22616 Programming With Python (PWP) Notes

6th Sem MCQ Test Series (All Subjects) : click here
6th Sem MCQ PDFs (All Subjects) : click here

Chapter No.	Name of chapter
1	Introduction and Syntax of Python Program
2	Python Operators and Control Flow Statements
3	Data Structures in Python
4	Python Functions, Modules and Packages
5	Object Oriented Programming in Python
6	File I/O Handling and Exception Handling

Course Outcome (CO): Display message on screen using Python Script on IDE.

Unit 1: Introduction and syntax of python programming

Python is developed by **Guido van Rossum**. Guido van Rossum started implementing Python in 1989.

Features of python –

Python is Interactive –

You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented –

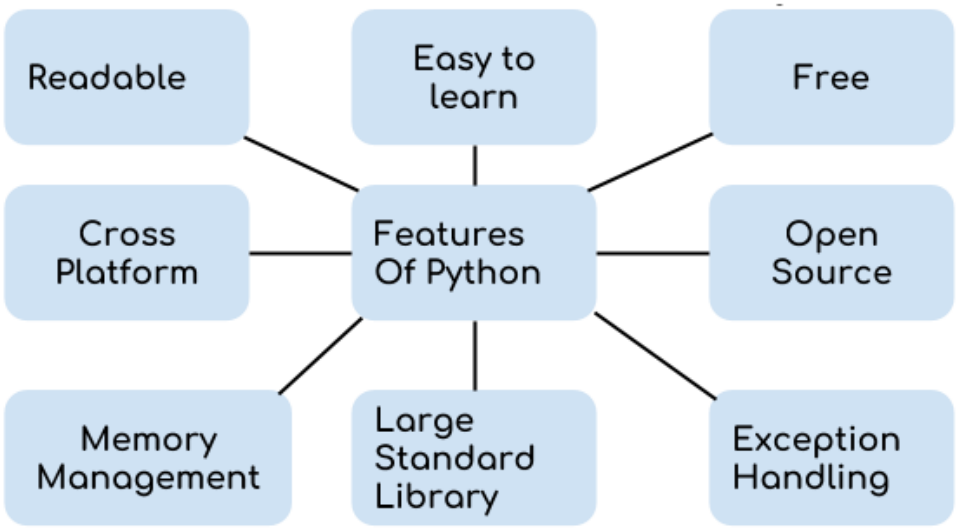
Python supports object oriented language and concepts of classes and objects come into existence.

Python is Interpreted

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

Python is Platform Independent

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.



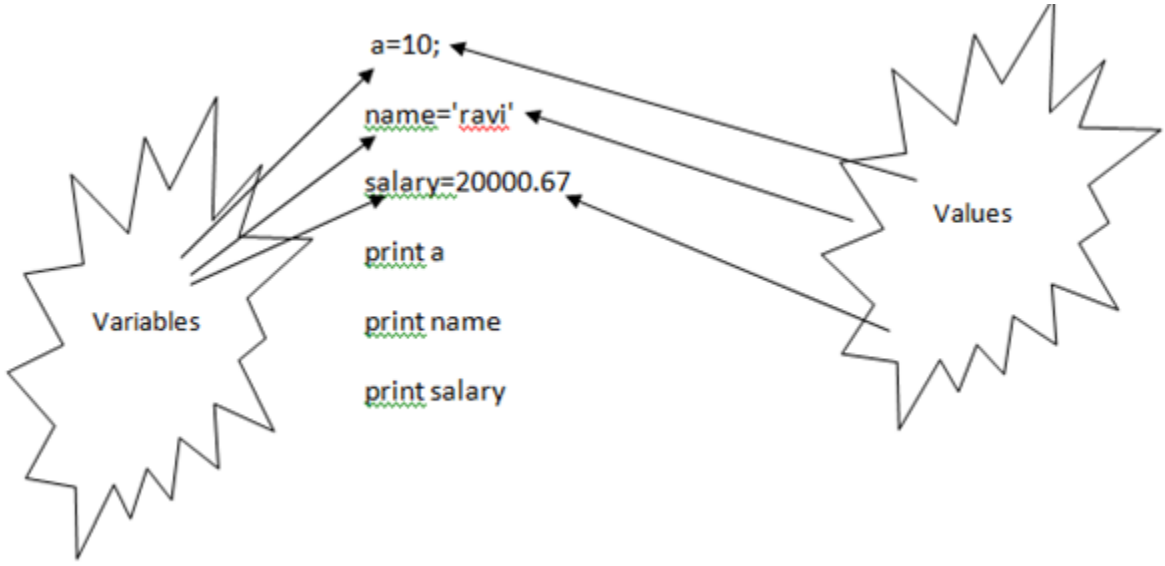
Python building blocks

Python Identifiers

Variable name is known as identifier.

The rules to name an identifier are given below.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore or digit (0-9).
- Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive for example my name, and MyName is not the same.
- Examples of valid identifiers : a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.



Assigning single value to multiple variables

Eg: x=y=z=50

Assigning multiple values to multiple variables:

Eg: a,b,c=5,10,15

Reserved Words

The following list shows the Python keywords. These are reserved words and cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is compulsory.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
   print "True"
else:
   print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

Course Outcome (CO): Display message on screen using Python Script on IDE.

Variable Types

Variables are used to store data, they take memory space based on the type of value we assigning to them. Creating variables in Python is simple, you just have write the variable name on the left side of = and the value on the right side.

Python Variable Example

```
num = 100
str = "BeginnersBook"
print(num)
print(str)
```

Multiple Assignment Examples

We can assign multiple variables in a single statement like this in Python.

```
x = y = z = 99
print(x)
print(y)
print(z)
```

Another example of multiple assignments

```
a, b, c = 5, 6, 7
print(a)
print(b)
print(c)
```

Plus and concatenation operation on the variables

```
x = 10
y = 20
print(x + y)

p = "Hello"
q = "World"
print(p + " " + q)
```

output:

30

Hello World

Comments

Use the hash (#) symbol to start writing a comment.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

use triple quotes, either ''' or """.

eg:

```
"""This is also a
perfect example of
multi-line comments"""
```

Data Types

A data type defines the type of data, for example 123 is an integer data while “hello” is a String type of data. The data types in Python are divided in two categories:
1. Immutable data types – Values cannot be changed.
2. Mutable data types – Values can be changed

Immutable data types in Python are:
1. Numbers
2. String
3. Tuple

Mutable data types in Python are:
1. List
2. Dictionaries
3. Sets

Python Environment Setup-Installation and Working Of IDE

Course Outcome (CO): Display message on screen using Python Script on IDE.

Install Python on any operating system such as Windows, Mac OS X, Linux/Unix and others.

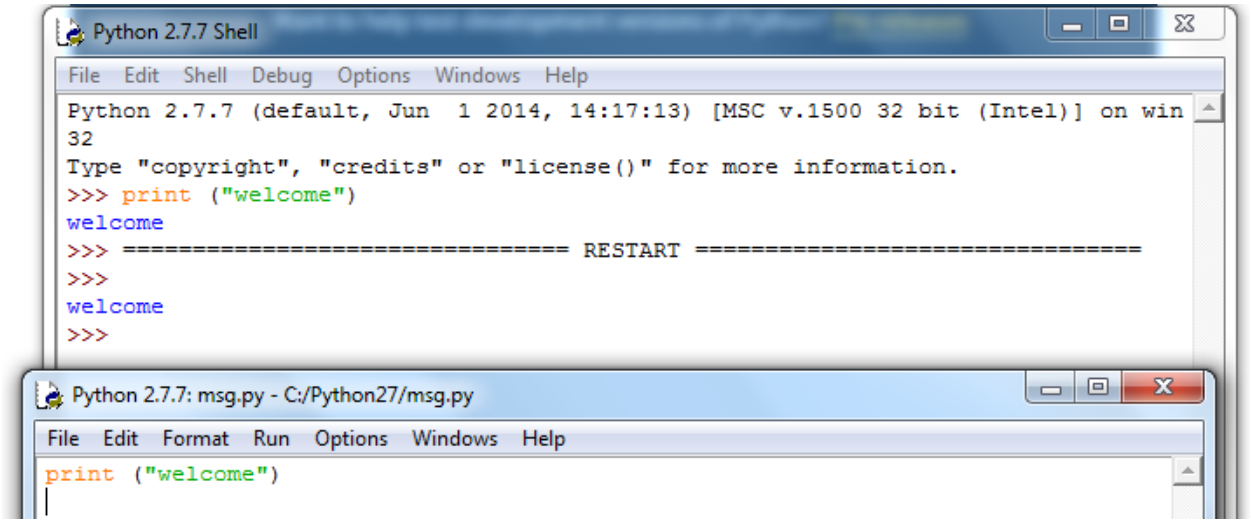
To install the Python on your operating system, go to this link: https://www.python.org/downloads/. You will see a screen like this.

1. On Windows 7 and earlier, IDLE is easy to start—it's always present after a Python install, and has an entry in the Start button menu for Python in Windows 7 and earlier.

2. Select it by right-clicking on a Python program icon, and launch it by clicking on the icon for the files idle.pyw or idle.py located in the idlelib subdirectory of Python's Lib directory. In this mode, IDLE is a clickable Python script that lives in C:\Python3.6\..

Running Simple Python Scripts To Display 'Welcome' Message

Course Outcome (CO): Display message on screen using Python Script on IDE.



Python data types : numbers ,string, tuples, lists, dictionary.

Python Data Types

1 Python Data Type – Numeric
2 Python Data Type – String
3 Python Data Type – List
4 Python Data Type – Tuple
5 Dictionary

Declaration and use of data types

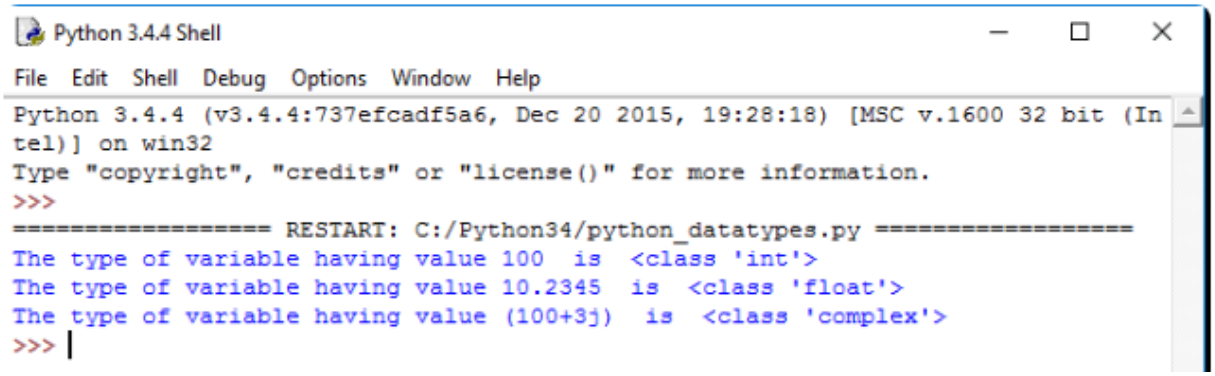
Python Data Type – Numeric

Python numeric data type is used to hold numeric values like;

- int – holds signed integers of non-limited length.
- long- holds long integers(exists in Python 2.x, deprecated in Python 3.x).
- float- holds floating precision numbers and it's accurate upto 15 decimal places.
- complex- holds complex numbers.

```
#create a variable with integer value.
a=100
print("The type of variable having value", a, " is ", type(a))

#create a variable with float value.
b=10.2345
print("The type of variable having value", b, " is ", type(b))

#create a variable with complex value.
c=100+3j
print("The type of variable having value", c, " is ", type(c))
```

If you run the above code you will see output like the below image.



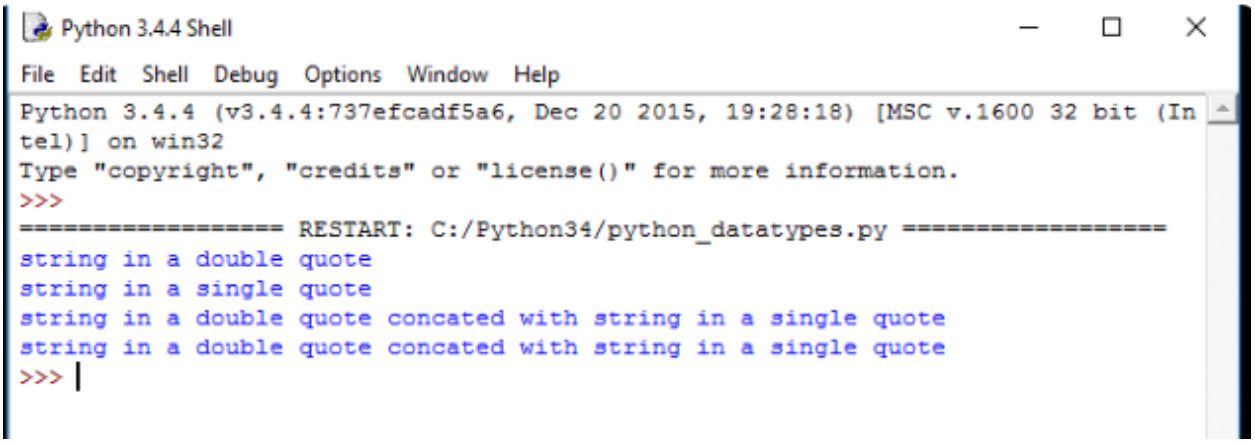
Python Data Type – String

The string is a sequence of characters. Python supports Unicode characters. Generally, strings are represented by either single or double quotes.

Course Outcome (CO): Display message on screen using Python Script on IDE.

```
a = "string in a double quote"
b= 'string in a single quote'
print(a)
print(b)

# using ',' to concatenate the two or several strings
print(a,"concatenated with",b)

#using '+' to concate the two or several strings
print(a+" concated with "+b)
```

The above code produce output like below picture-

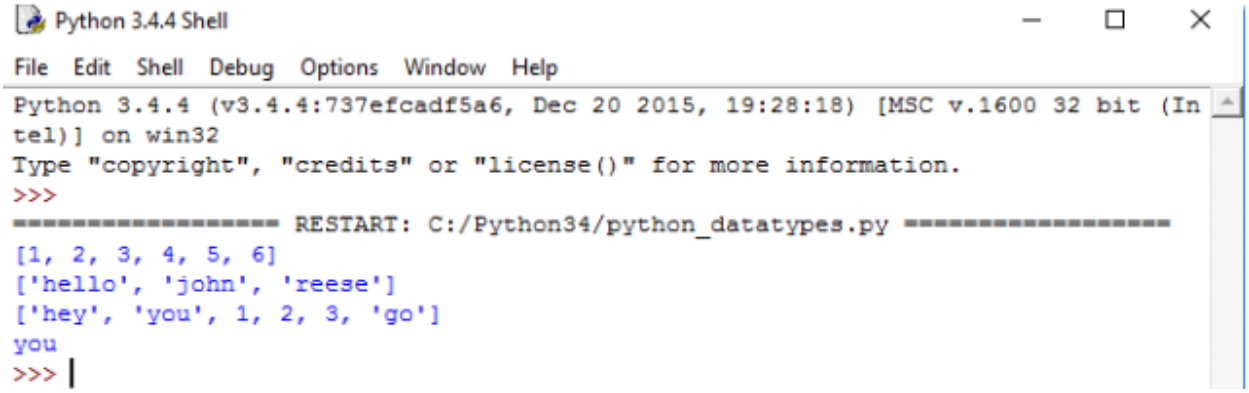


Python Data Type – List

List is an ordered sequence of some data written using square brackets ([]) and commas (,).

```
#list of having only integers
a= [1,2,3,4,5,6]
print(a)

#list of having only strings
b=["hello","john","reese"]
print(b)

#list of having both integers and strings
c= ["hey","you",1,2,3,"go"]
print(c)

#index are 0 based. this will print a single character
print(c[1]) #this will print "you" in list c
```

The above code will produce output like this-



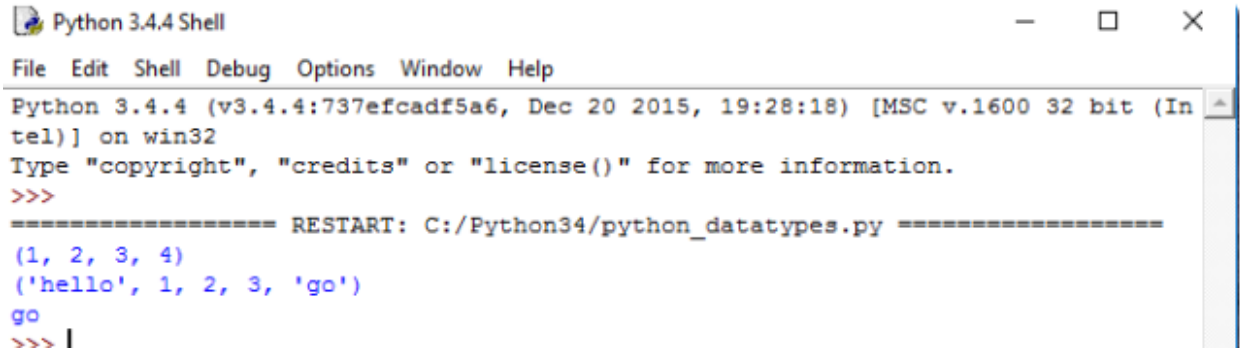
Python Data Type – Tuple

Tuple is another data type which is a sequence of data similar to list. But it is immutable. That means data in a tuple is write protected. Data in a tuple is written using parenthesis and commas.

Course Outcome (CO): Display message on screen using Python Script on IDE.

```
#tuple having only integer type of data.
a=(1,2,3,4)
print(a) #prints the whole tuple

#tuple having multiple type of data.
b=("hello", 1,2,3,"go")
print(b) #prints the whole tuple

#index of tuples are also 0 based.

print(b[4])
```

The output of this above python data type tuple example code will be like below image.



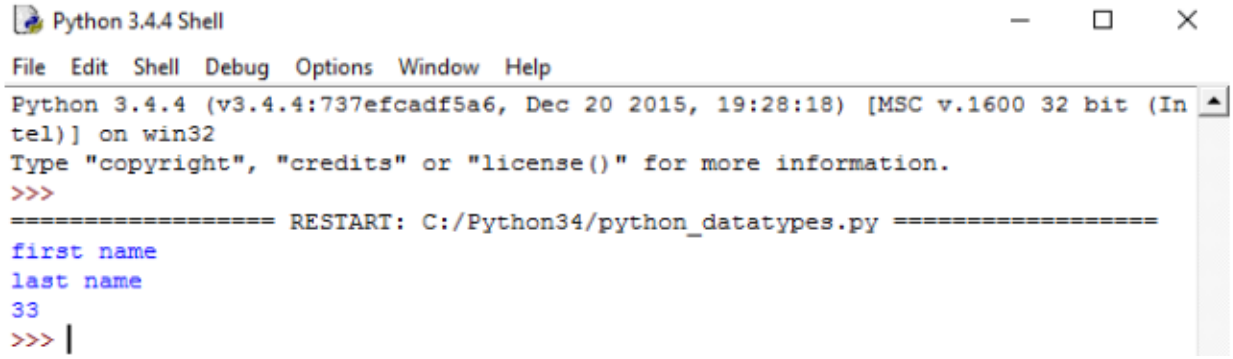
Dictionary

Python Dictionary is an unordered sequence of data of key-value pair form. It is similar to the hash table type. Dictionaries are written within curly braces in the form **key: value.**

Course Outcome (CO): Display message on screen using Python Script on IDE.

```
#a sample dictionary variable

a = {1:"first name",2:"last name", "age":33}

#print value having key=1
print(a[1])
#print value having key=2
print(a[2])
#print value having key="age"
print(a["age"])
```

If you run this python dictionary data type example code, output will be like below image.



UNIT 2: PYTHON OPERATORS AND CONTROL FLOW STATEMENTS

BASIC OPERATORS:

- **Arithmetic operators**

  +(addition) - (subtraction) *(multiplication)

  /(divide) %(reminder) //(floor division) exponent (**)

- **Comparison operators**

  == != <= >= > <

- **Assignment Operators**

  = += -= *= %= **= //=

- **Logical Operators**

  and or not

- **Bitwise Operators**

  & (binary and) | (binary or) ^ (binary xor)

  << (left shift) >> (right shift) ~ (negation)

- **Membership Operators**

  in not in

- **Identity Operators**

  is is not

CO: DEVELOP PYTHON PROGRAM TO DEMONSTRATE USE OF OPERATORS

1. Arithmetic operators

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a - b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

2. Comparison/Relational operators

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.

>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

3. Assignment operator

Operator	Description
=	It assigns the the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

4. Logical operators

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression **a** is true then not (a) will be false and vice versa.

5. Bitwise operators

The bitwise operators perform bit by bit operation on the values of the two operands.

For example,

```
if a = 7;
  b = 6;
then, binary (a) = 0111
   binary (b) = 0011

hence, a & b = 0011
    a | b = 0111
        a ^ b = 0100
    ~ a = 1000
```

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.

~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

6. Membership operators

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

7. Identity Operators

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.

8. Python Operator Precedence

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

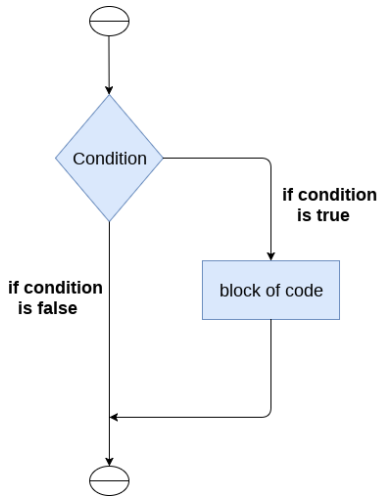
Operator	Description
**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus and minus.
* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus and minus
>> <<	Left shift and right shift
&	Binary and.
^ |	Binary xor and or
<= < > >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

2.2 control flow:

2.3conditional statements (if,if.....else,nested if)

The if statement

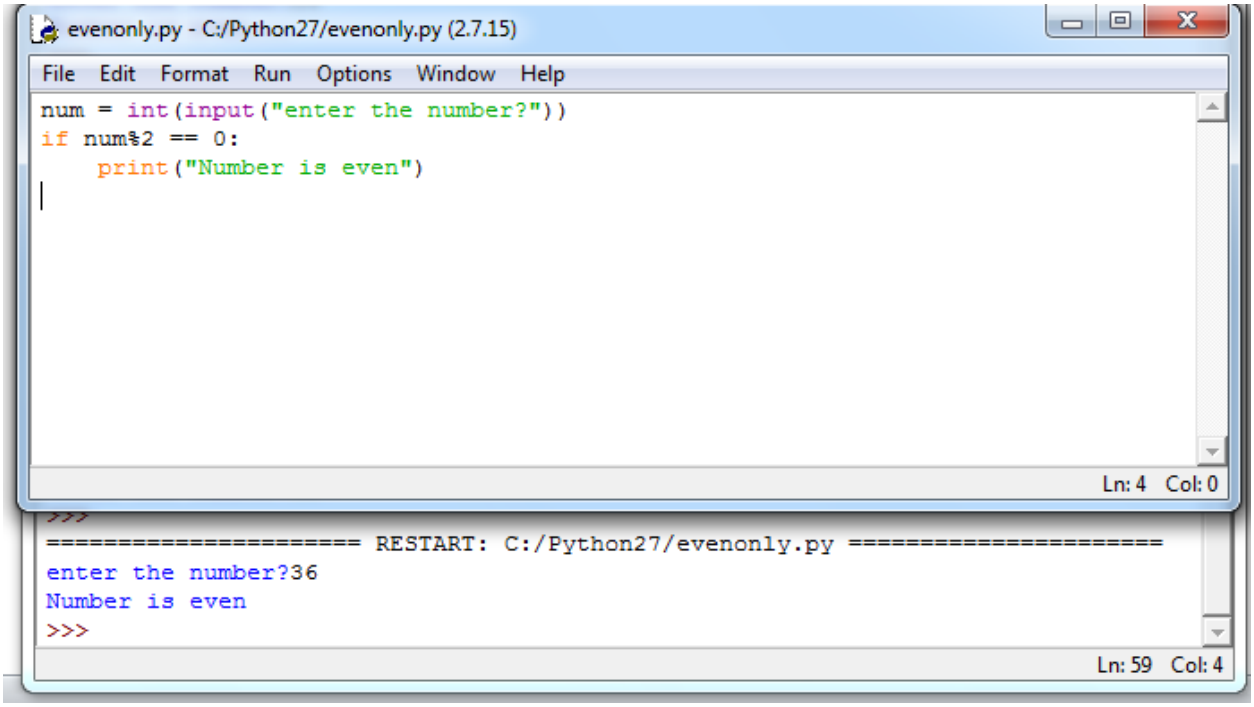
The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



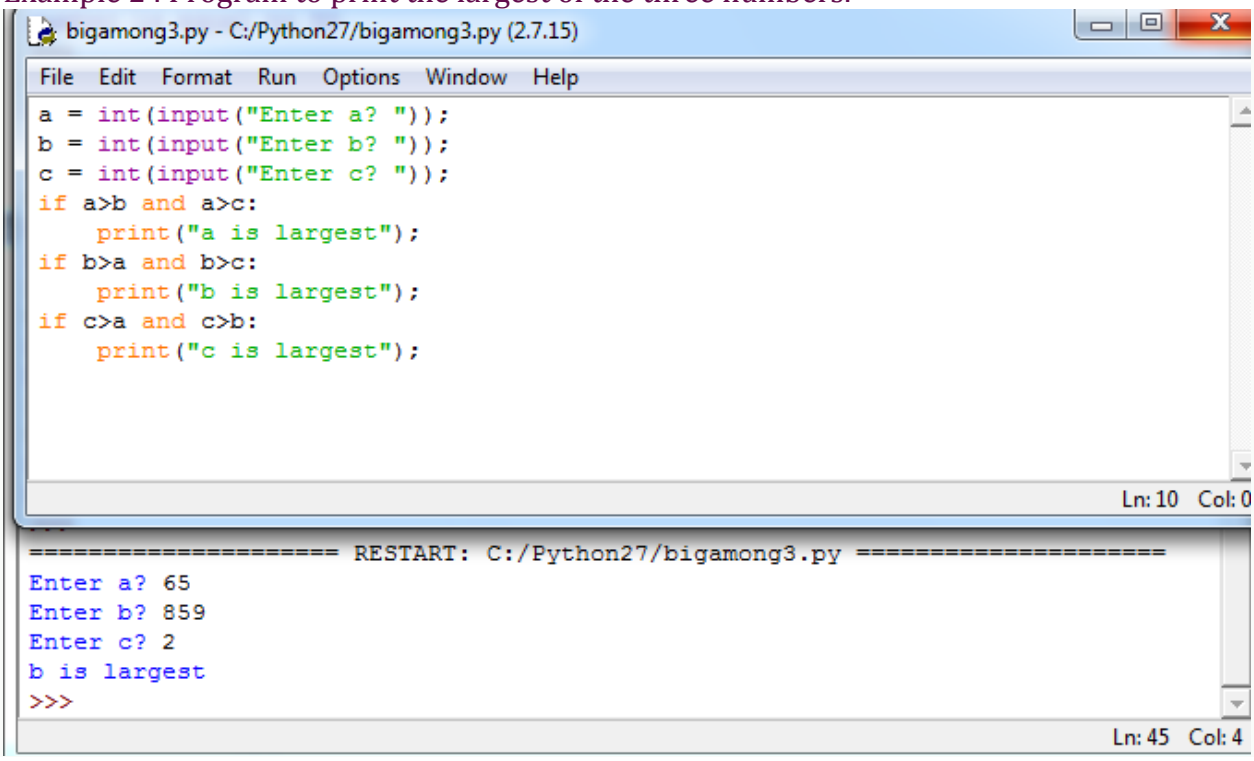
The syntax of the if-statement is given below.

```
if expression:
   statement
```

Example



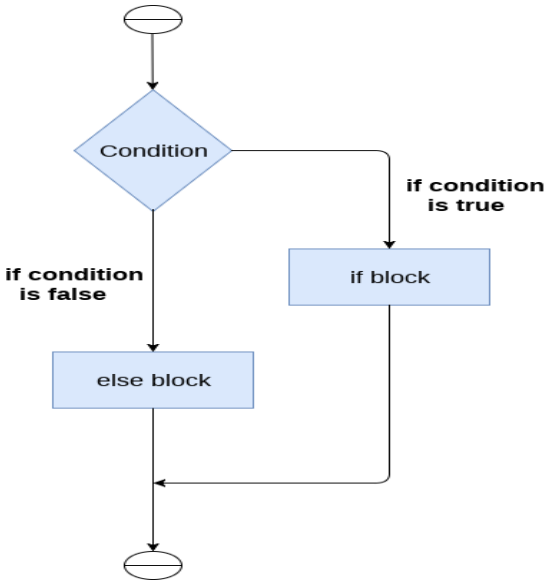
Example 2 : Program to print the largest of the three numbers.



The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

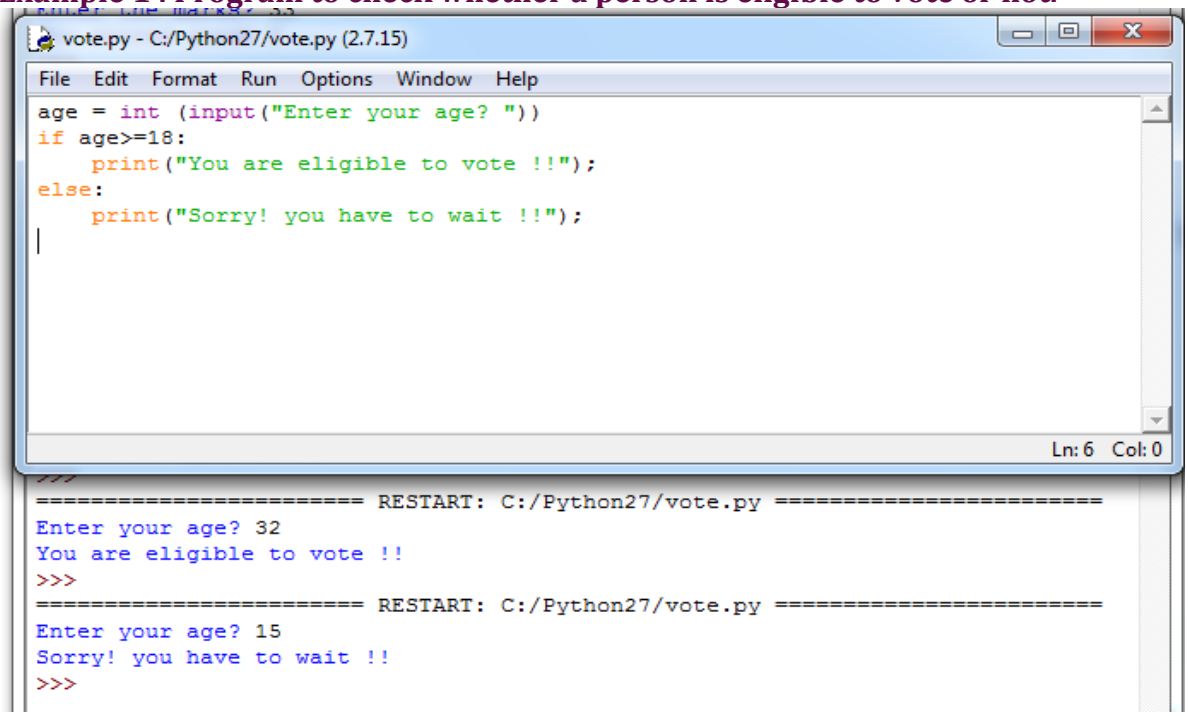
If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



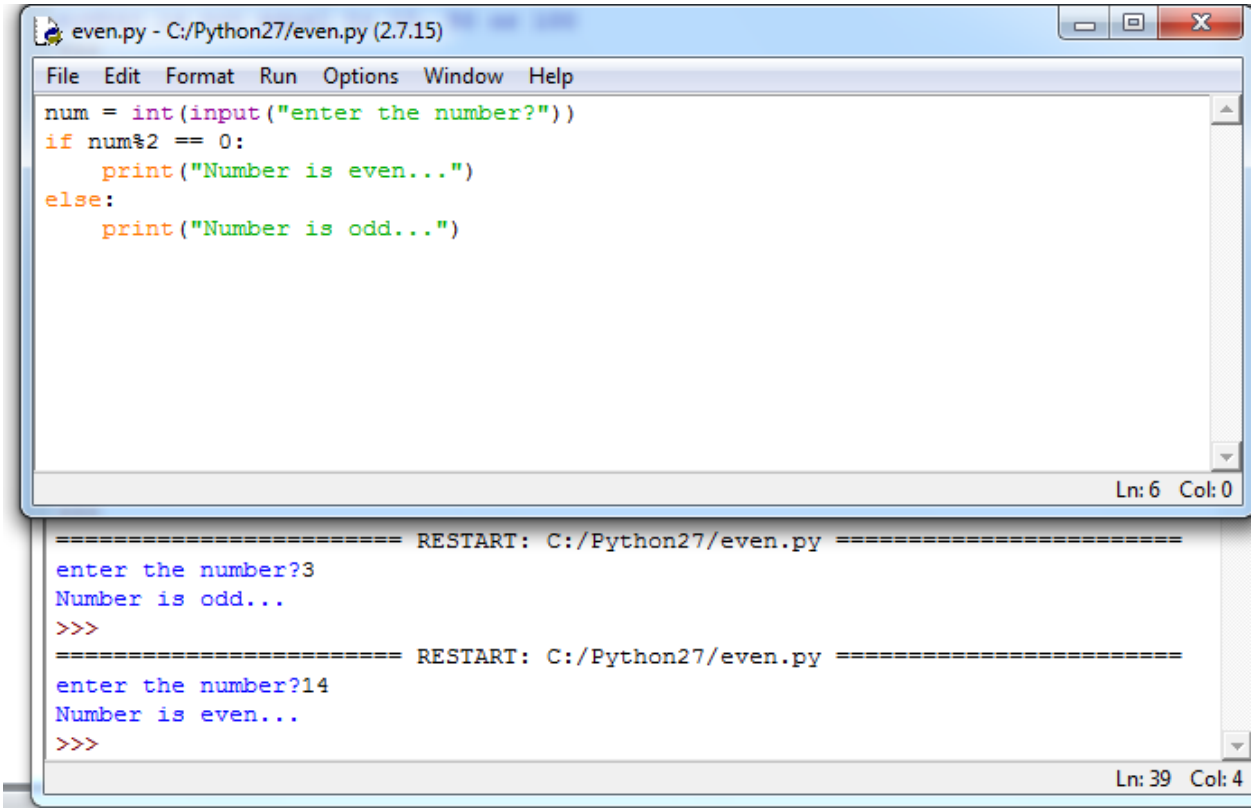
The syntax of the if-else statement is given below.

```
if condition:
   #block of statements
else:
   #another block of statements (else-block)
```

Example 1 : Program to check whether a person is eligible to vote or not.



Example 2: Program to check whether a number is even or not.



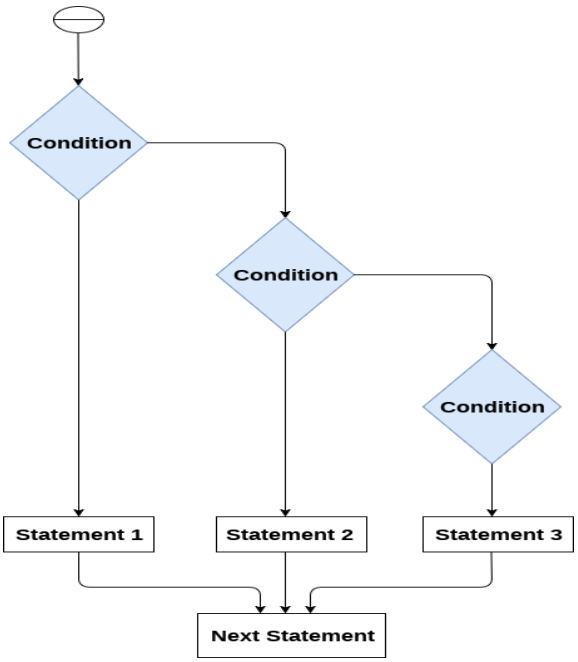
The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.

However, using elif is optional.

The syntax of the elif statement is given below.

```
if expression 1:
   # block of statements

elif expression 2:
   # block of statements

elif expression 3:
   # block of statements

else:
   # block of statements
```



Example 1

elif1.py - C:/Python27/elif1.py (2.7.15)

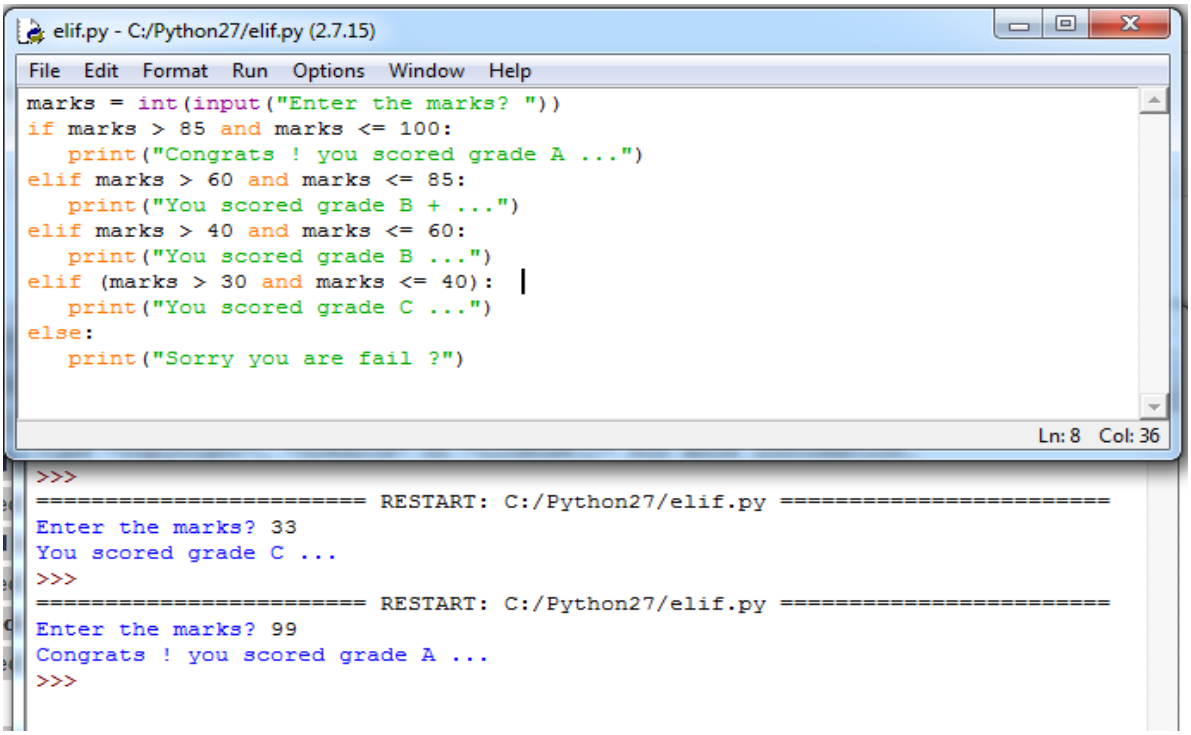
File Edit Format Run Options Window Help

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Ln: 10 Col: 0

```
========== RESTART: C:/Python27/elif1.py ==========
Enter the number?36
number is not equal to 10, 50 or 100
>>>
========== RESTART: C:/Python27/elif1.py ==========
Enter the number?20
number is not equal to 10, 50 or 100
>>>
========== RESTART: C:/Python27/elif1.py ==========
Enter the number?50
number is equal to 50
>>>
```

Example 2



2.4looping in python (while loop,for loop,nested loops)

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

Loop Statement	Description
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

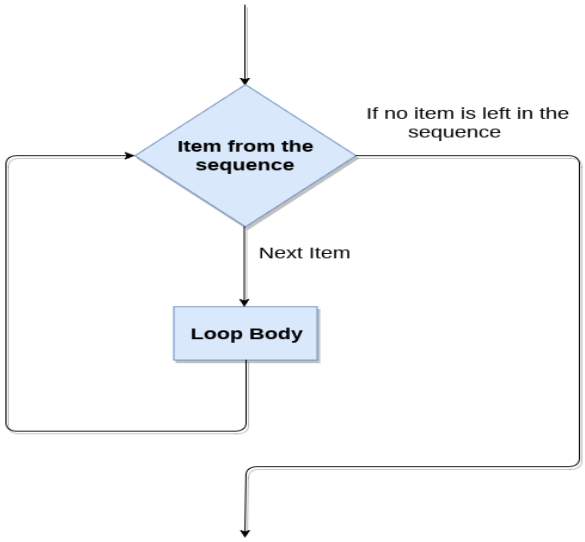
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
do-while loop	The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

Python for loop

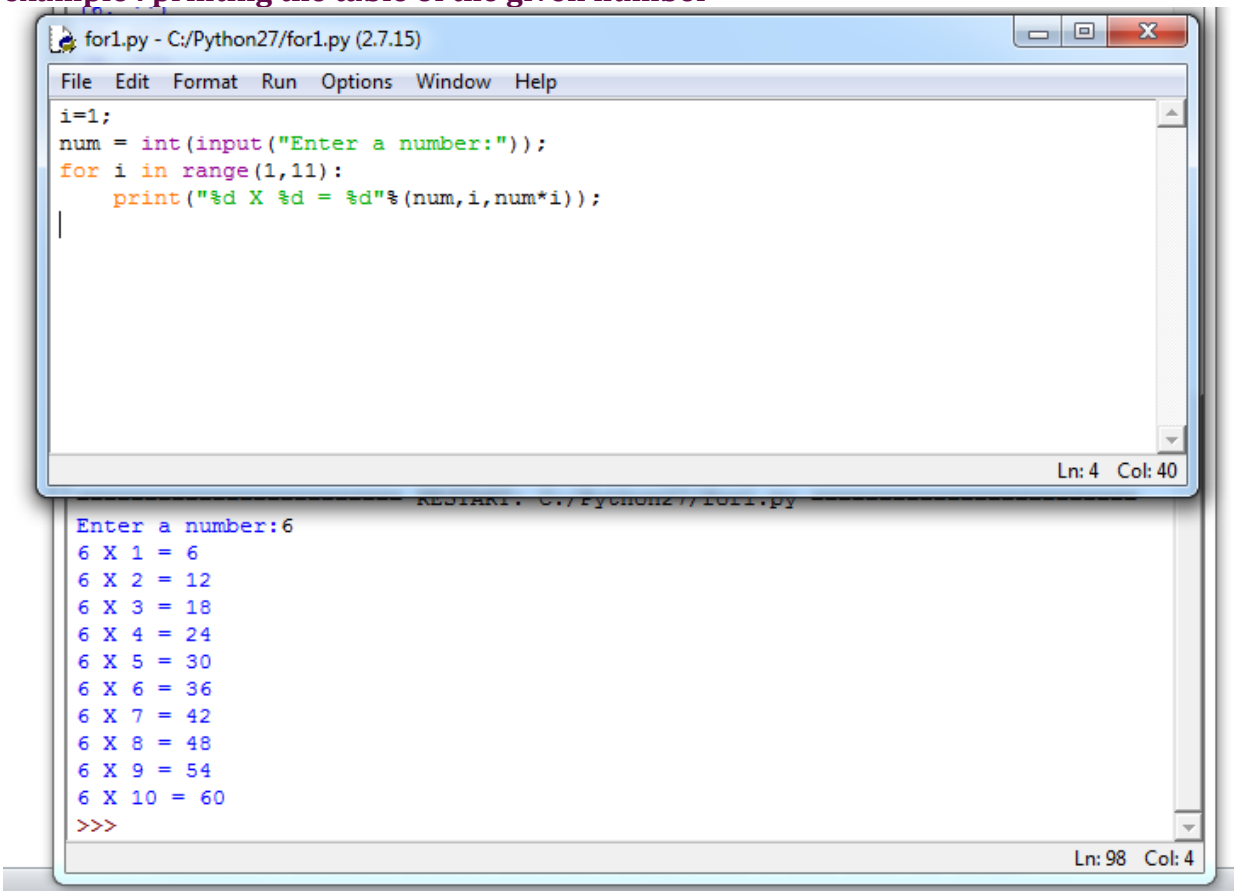
The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

```
for iterating_var in sequence:
    statement(s)
```



example : printing the table of the given number



Nested for loop in python

Python allows us to nest any number of for loops inside a for loop.

The inner loop is executed n number of times for every iteration of the outer loop.

The syntax of the nested for loop in python is given below.

```
for iterating_var1 in sequence:
   for iterating_var2 in sequence:
      #block of statements
#Other statements
```

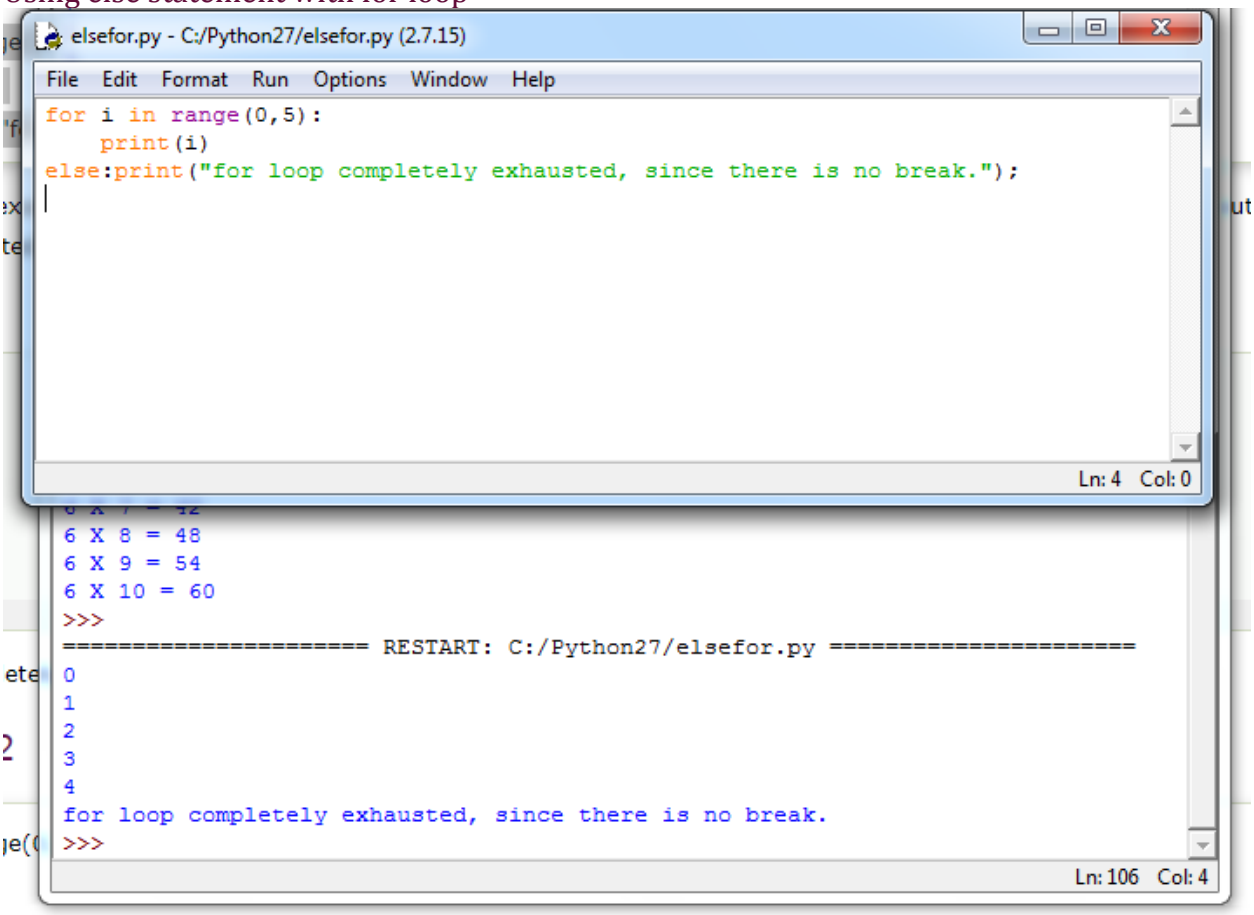
Example 1

```
n = int(input("Enter the number of rows you want to print?"))
i,j=0,0
for i in range(0,n):
   print()
   for j in range(0,i+1):
      print("*",end="")
```

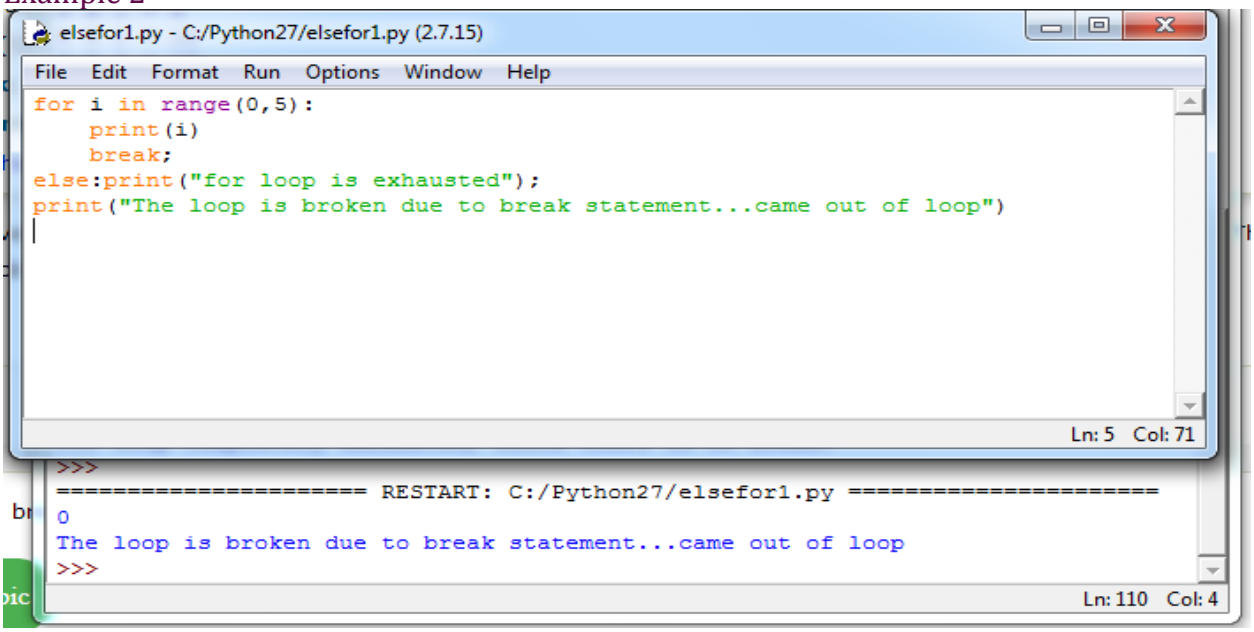
Output:

```
Enter the number of rows you want to print?5
*
**
***
****
```

Using else statement with for loop



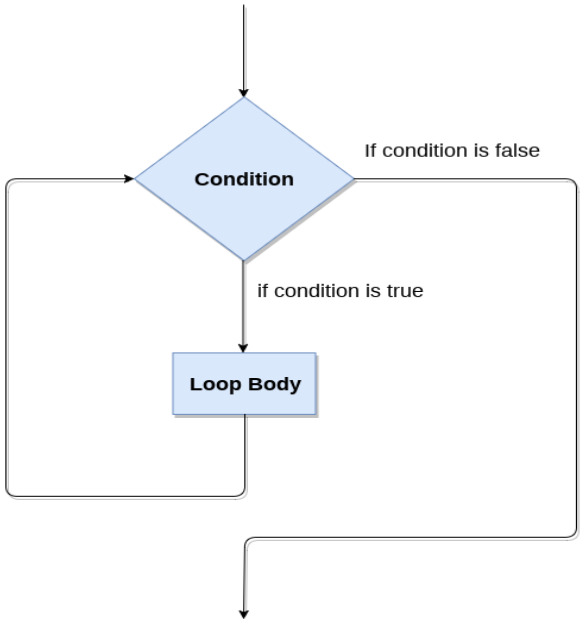
Example 2



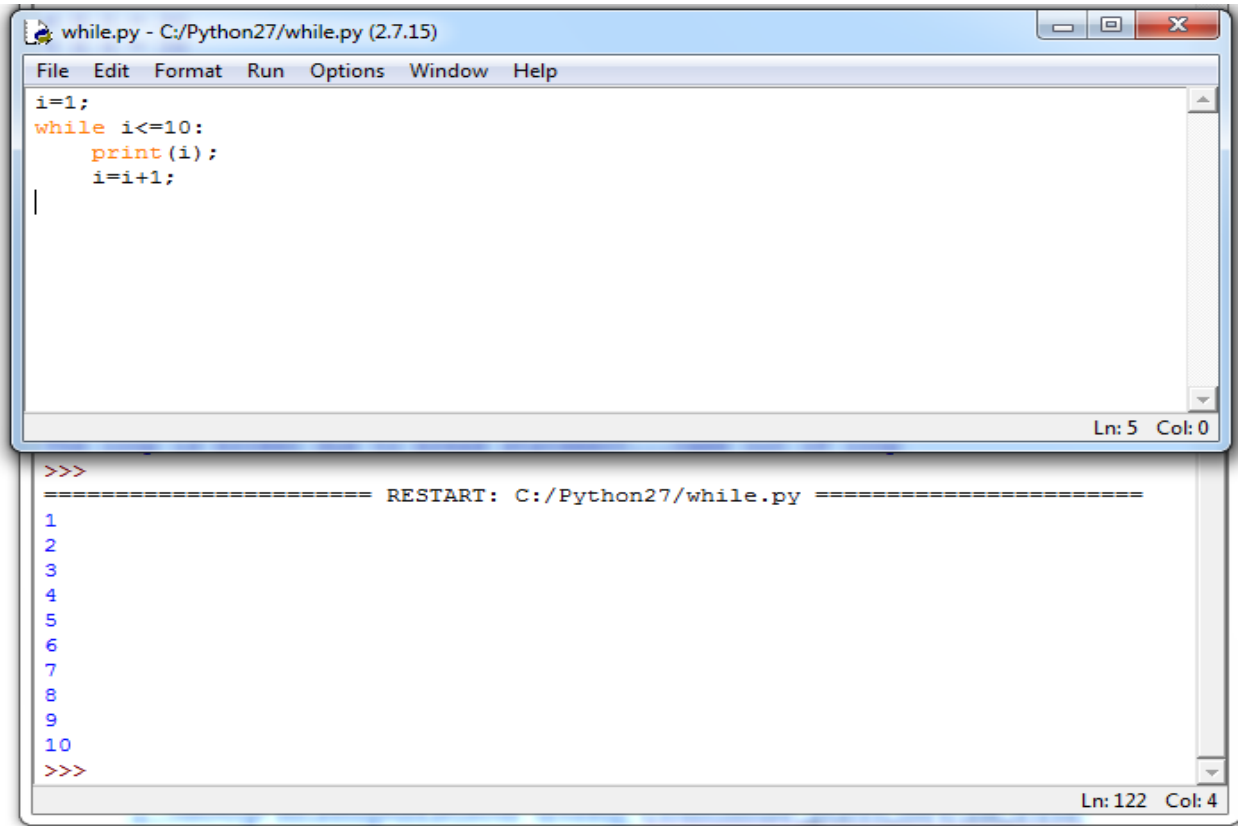
Python while loop

while expression:

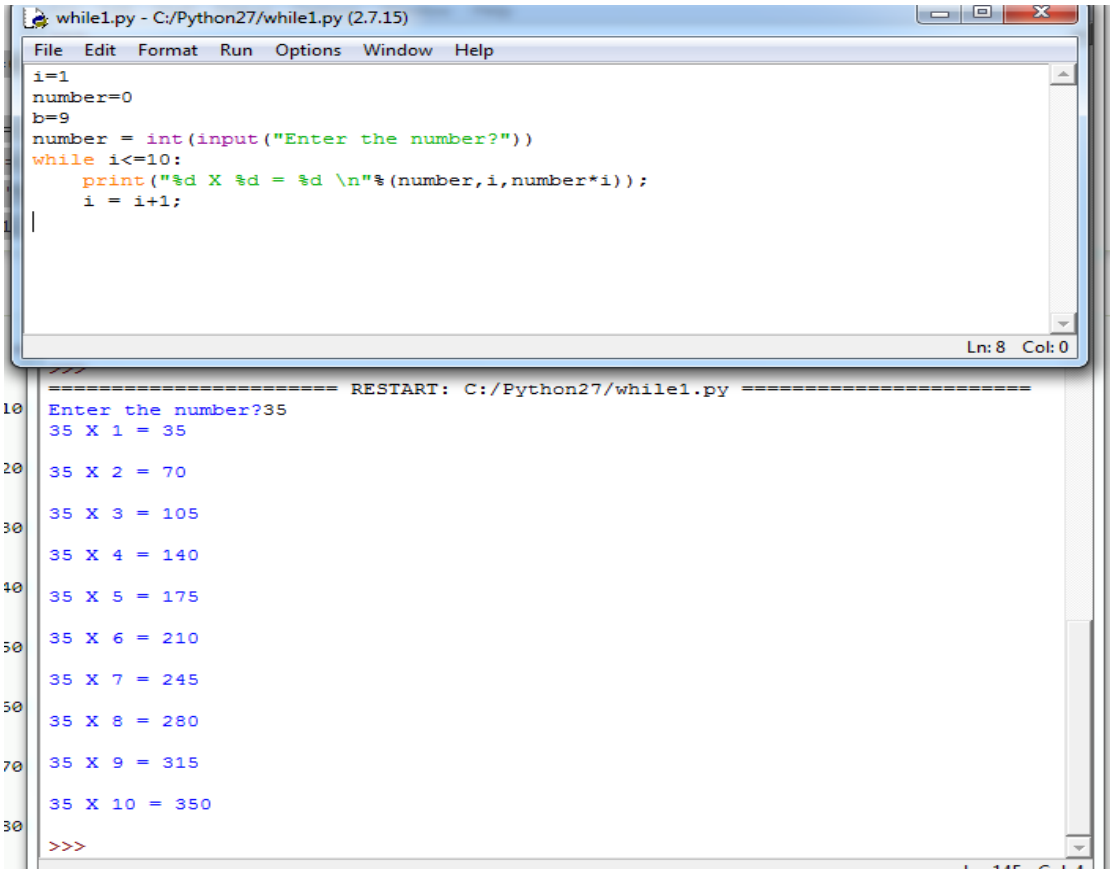
statements



Example 1


while.py - C:/Python27/while.py (2.7.15)
File Edit Format Run Options Window Help
i=1;
while i<=10:
print(i);
i=i+1;
Ln: 5 Col: 0
>>>
========== RESTART: C:/Python27/while.py ==========
1
2
3
4
5
6
7
8
9
10
>>>
Ln: 122 Col: 4

Example 2:

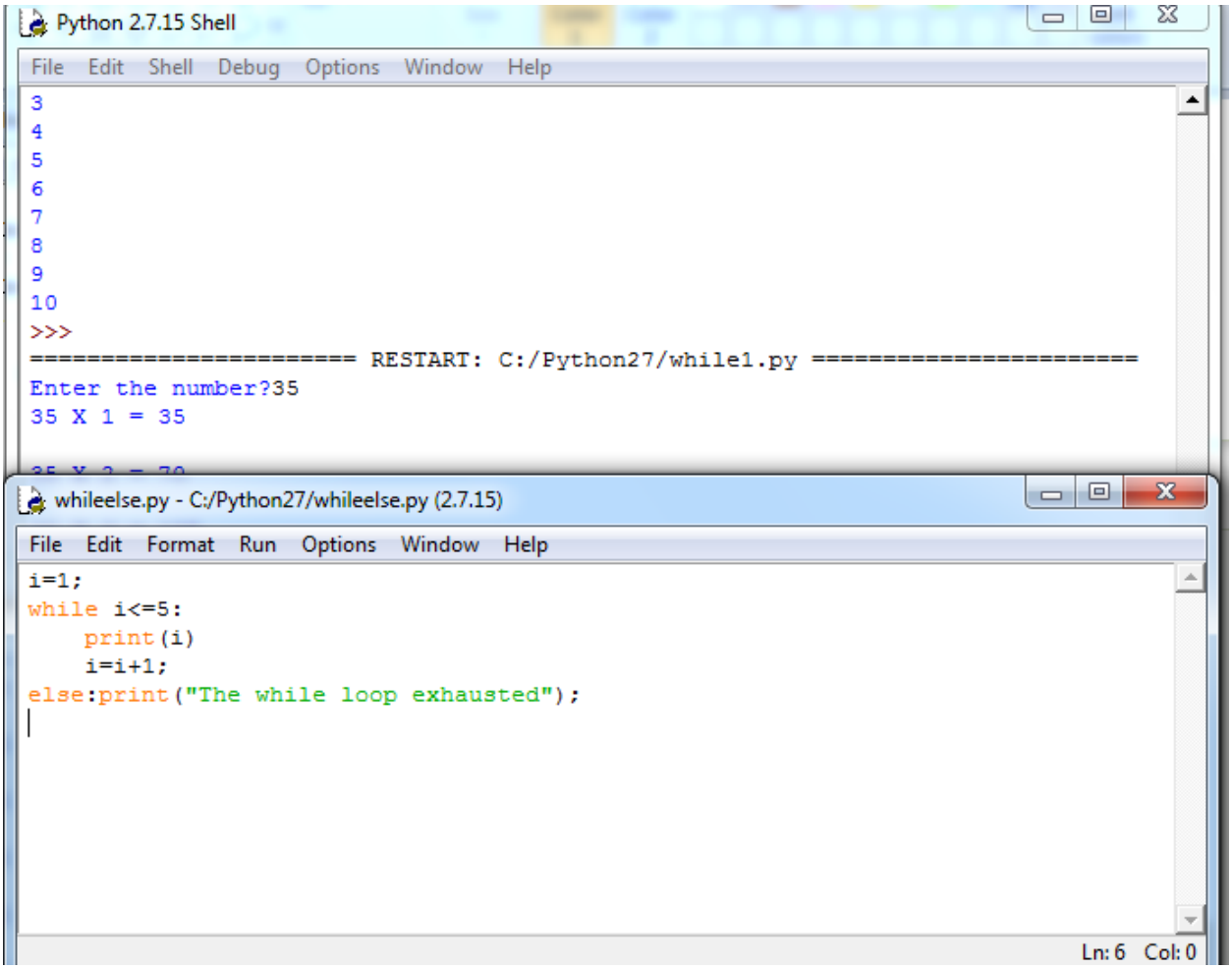

while1.py - C:/Python27/while1.py (2.7.15)
File Edit Format Run Options Window Help
i=1
number=0
b=9
number = int(input("Enter the number?"))
while i<=10:
print("%d X %d = %d \n"%(number,i,number*i));
i = i+1;
Ln: 8 Col: 0
========== RESTART: C:/Python27/while1.py ==========
Enter the number?35
35 X 1 = 35
35 X 2 = 70
35 X 3 = 105
35 X 4 = 140
35 X 5 = 175
35 X 6 = 210
35 X 7 = 245
35 X 8 = 280
35 X 9 = 315
35 X 10 = 350
>>>

Using else with Python while loop

The else block is executed when the condition given in the while statement becomes false.

if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Consider the following example.



2.5loop manipulation using continue, pass, break, else

Python continue Statement

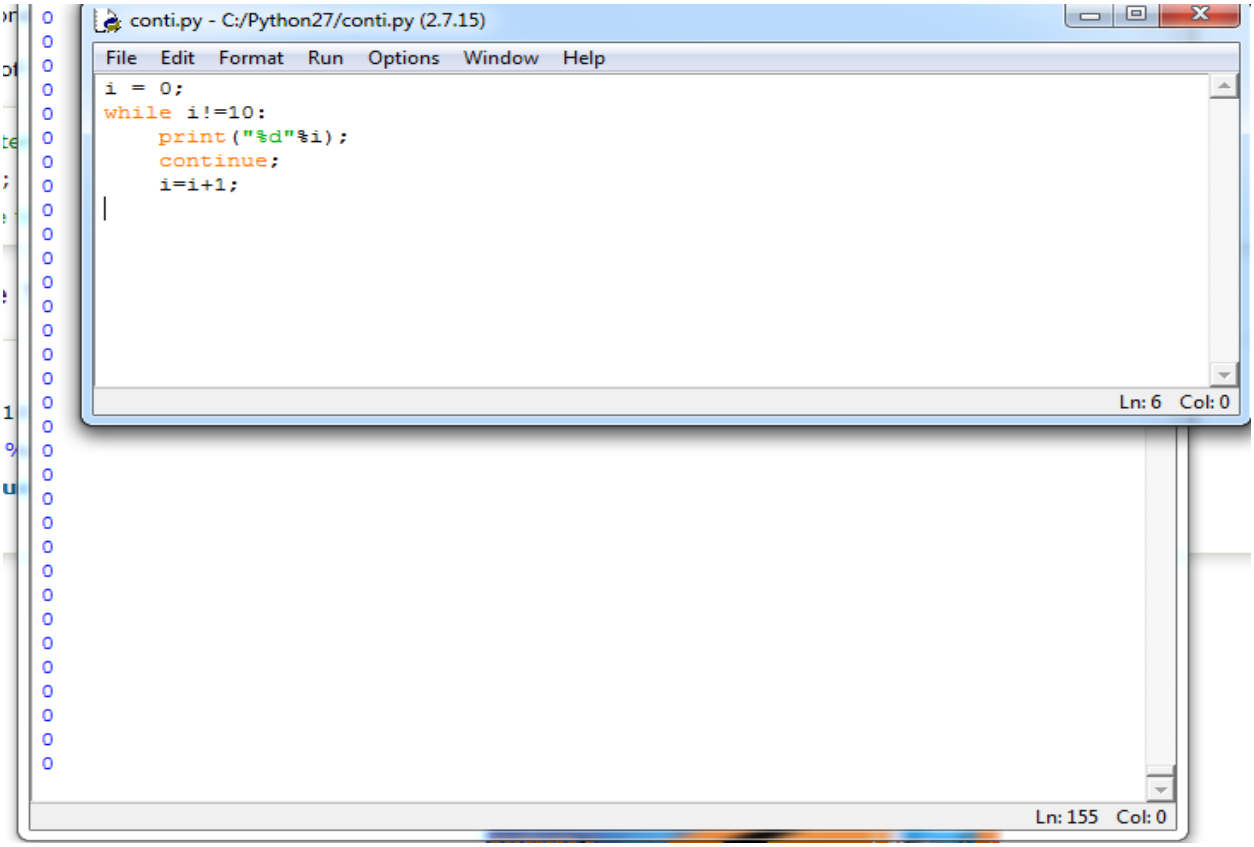
The continue statement in python is used to bring the program control to the beginning of the loop.

The continue statement skips the remaining lines of code inside the loop and start with the next iteration.

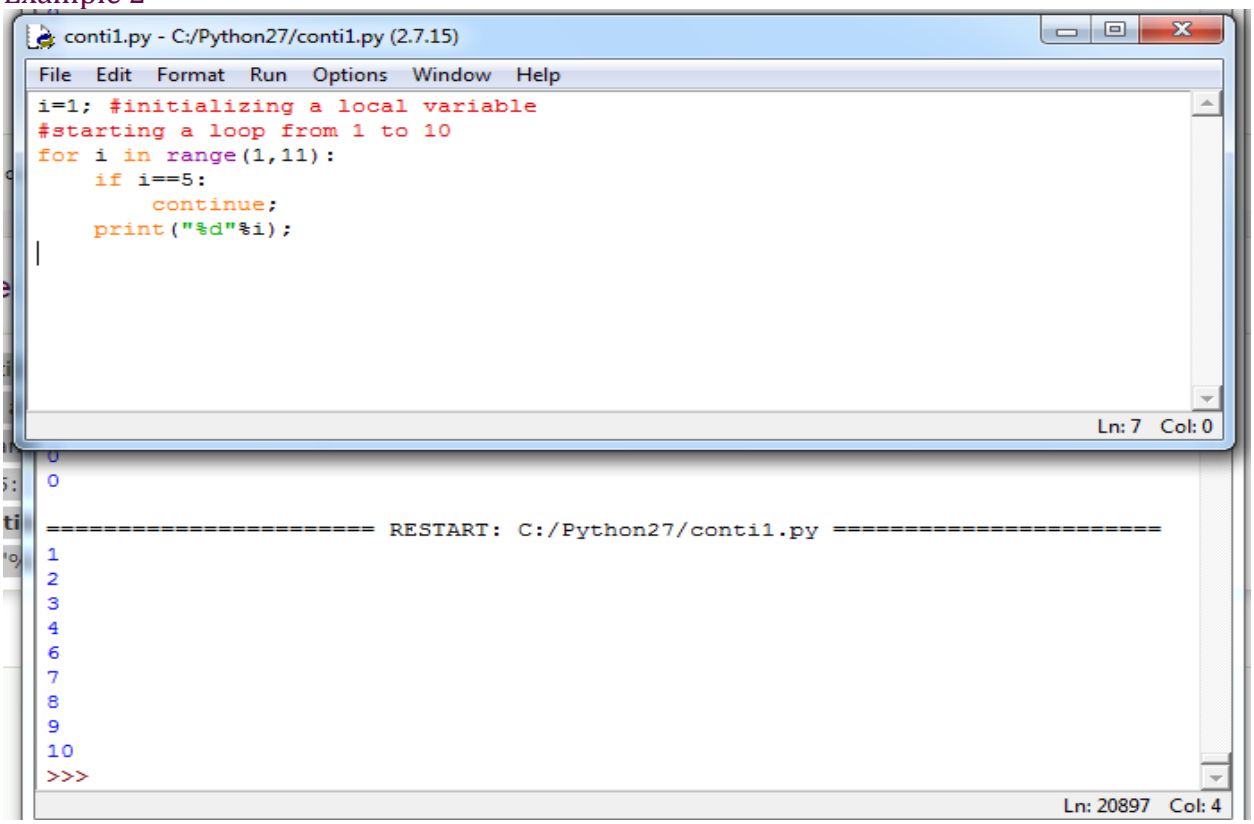
It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

The syntax of Python continue statement is given below.

```
#loop statements
continue;
#the code to be skipped
```



Example 2



Python Pass

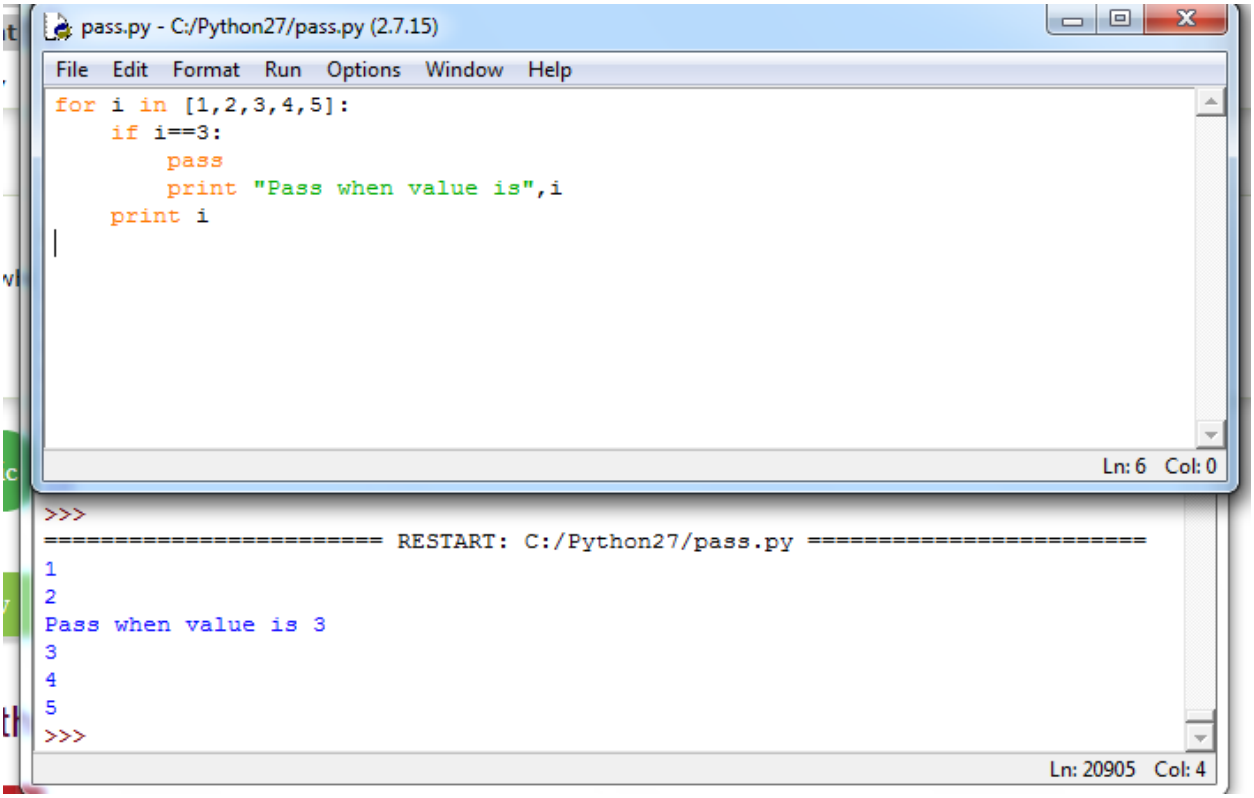
In Python, pass keyword is used to execute nothing;

It means, when we don't want to execute code, the pass can be used to execute empty.

It is same as the name refers to.

Python Pass Syntax

```
pass
```



Python break statement

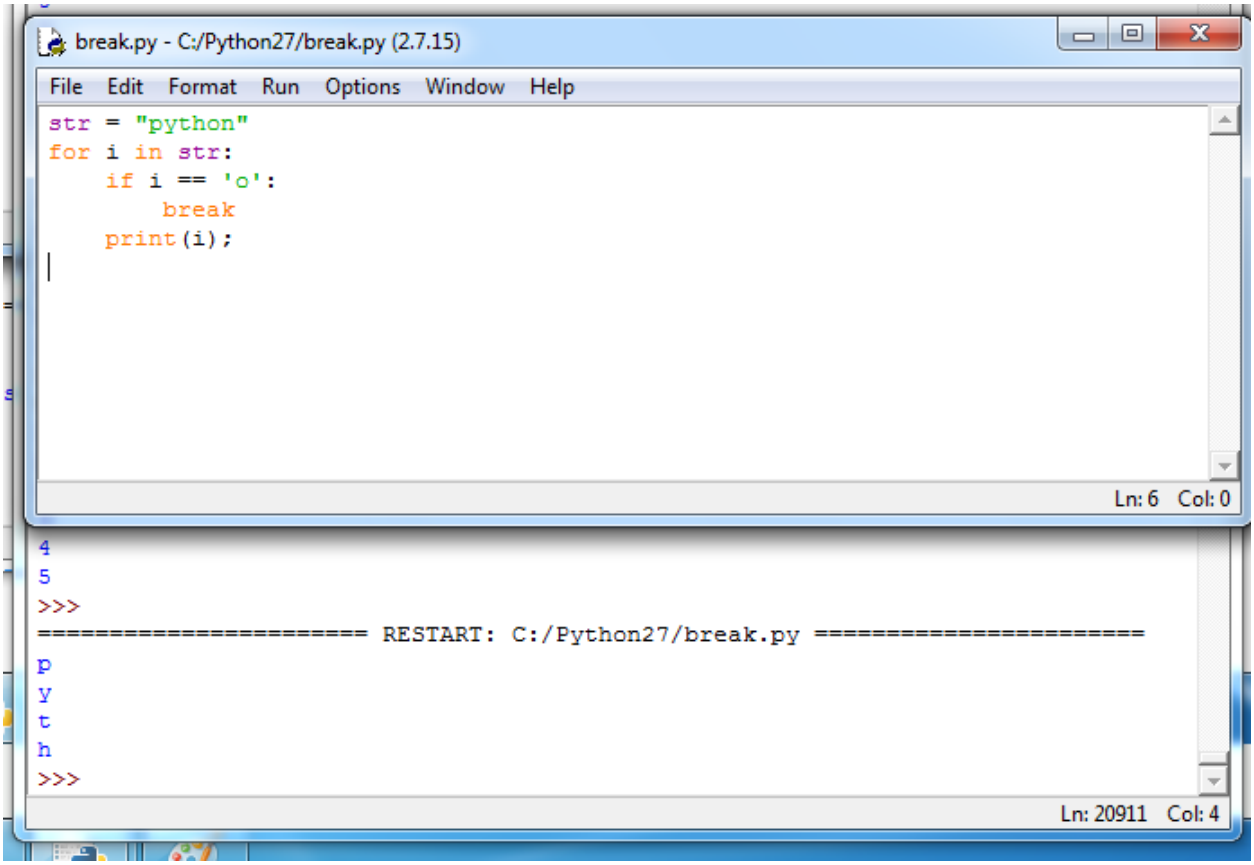
The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.

break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

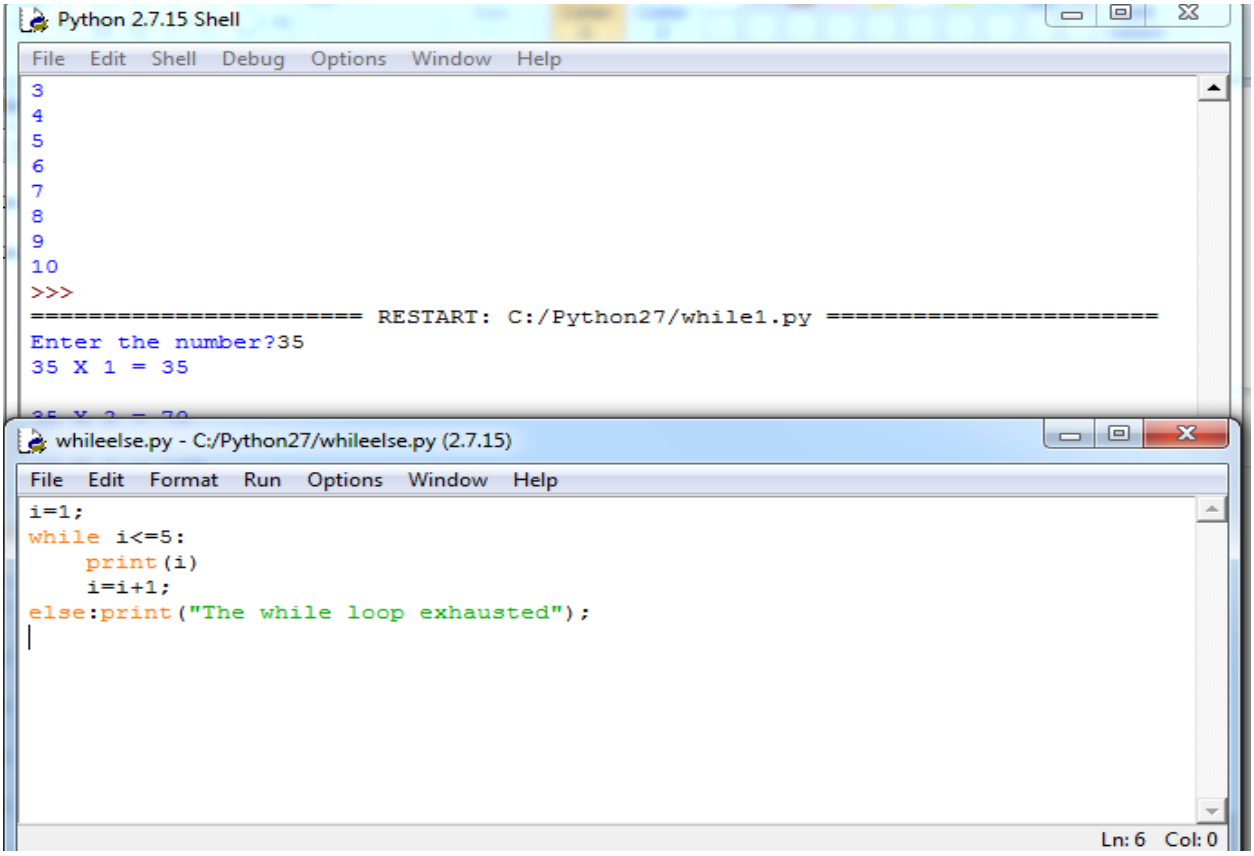
The syntax of the break is given below.

```
#loop statements
break;
```


break.py - C:/Python27/break.py (2.7.15)
File Edit Format Run Options Window Help
str = "python"
for i in str:
if i == 'o':
break
print(i);
Ln: 6 Col: 0
4
5
>>>
========== RESTART: C:/Python27/break.py ==========
p
y
t
h
>>>
Ln: 20911 Col: 4

Python else statement

The else block is executed when the condition given in the loop statement becomes false.



UNIT 3: DATA STRUCTURES IN PYTHON

Python List

- List in python is implemented to store the sequence of various type of data
- A list can be defined as a collection of values or items of different types.
- The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be defined as follows.

1. L1 = ["MMP", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]
3. L3 = [1, "GAD"]

Accessing List

The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1.

The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

Consider the following example.

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

List[2:4] = [2, 3]

List[1:3] = [1, 2]

List[:4] = [0, 1, 2, 3]

Updating List values

Lists are the most versatile data structures in python since they are immutable and their values can be updated by using the slice and assignment operator.

List = [1, 2, 3, 4, 5, 6]

```
print(List)
```

Output :[1, 2, 3, 4, 5, 6]

```
List[2] = 10;
print(List)
```

Output: [1, 2, 10, 4, 5, 6]

```
List[1:3] = [89, 78]
print(List)
```

Output : [1, 89, 78, 4, 5, 6]

Deleting List values

The list elements can also be deleted by using the **del** keyword. Python also provides us the remove() method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

```
List = [0,1,2,3,4]
print(List)
```

Output:

```
[0, 1, 2, 3, 4]
```

```
del List[0]
print(List)
```

Output:

```
[1, 2, 3, 4]
```

```
del List[3]
print(List)
```

Output:

```
[1, 2, 3]
```

Python List Built-in functions

Python provides the following built-in functions which can be used with the lists.

SN	Function	Description
1	cmp(list1, list2)	It compares the elements of both the lists.
2	len(list)	It is used to calculate the length of the list.
3	max(list)	It returns the maximum element of the list.
4	min(list)	It returns the minimum element of the list.
5	list(seq)	It converts any sequence to the list.

Python List Operations

The concatenation (+) and repetition (*) operator work in the same way as they were working with the strings.

Consider a List l1 = [1, 2, 3, 4] **and** l2 = [5, 6, 7, 8]

Operator	Description	Example
Repetition	The repetition operator enables the list elements to be repeated multiple times.	L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]
Concatenation	It concatenates the list mentioned on either side of the operator.	l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]
Membership	It returns true if a particular item exists in a particular list otherwise false.	print(2 in l1) prints True.
Iteration	The for loop is used to iterate over the list elements.	for i in l1:

		print(i) **Output** 1 2 3 4
Length	It is used to get the length of the list	len(l1) = 4

Python Tuple

- Python Tuple is used to store the sequence of immutable python objects.
- Tuple is immutable and the value of the items stored in the tuple cannot be changed.
- A tuple can be written as the collection of comma-separated values enclosed with the small brackets.

Where use tuple

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.

2. Tuple can simulate dictionary without keys. Consider the following nested structure which can be used as a dictionary.

[(101, "CO", 22), (102, "ME", 28), (103, "AE", 30)]

3. Tuple can be used as the key inside dictionary due to its immutable nature.

A tuple can be defined as follows.

```
T1 = (101, "Ayush", 22)
T2 = ("Apple", "Banana", "Orange")
```

Accessing tuple

The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the slice operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Tuple = (0, 1, 2, 3, 4, 5)

0	1	2	3	4	5

Tuple[0] = 0

Tuple[1] = 1

Tuple[2] = 2

Tuple[3] = 3

Tuple[4] = 4

Tuple[5] = 5

Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2:4] = (2, 3)

Tuple[1:3] = (1, 2)

Tuple[:4] = (0, 1, 2, 3)

The tuple items can not be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the del keyword with the tuple name

Consider the following example.

```
tuple1 = (1, 2, 3, 4, 5, 6)
print(tuple1)
del tuple1
print(tuple1)
```

Output:

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

Basic Tuple operations

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

Concatenation	It concatenates the tuple mentioned on either side of the operator.	T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)
Membership	It returns true if a particular item exists in the tuple otherwise false.	print (2 in T1) prints True.
Iteration	The for loop is used to iterate over the tuple elements.	for i in T1: print(i) **Output** 1 2 3 4 5
Length	It is used to get the length of the tuple.	len(T1) = 5

Python Tuple inbuilt functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

List VS Tuple

SN	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
2	The List is mutable.	The tuple is immutable.
3	The List has the variable length.	The tuple has the fixed length.
4	The list provides more functionality than tuple.	The tuple provides less functionality than the list.
5	The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary.

Python Set

Unordered collection of various items enclosed within the curly braces.

The elements of the set can not be duplicate.

The elements of the python set must be immutable.

Creating a set

Example 1: using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
print(type(Days))
```

Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}
<class 'set'>
```

Example 2: using set() method

```
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
print(Days)
```

Output:

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}
```

Accessing set values

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
print("the set elements ... ")
for i in Days:
  print(i)
```

Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}
the set elements ...
Friday
Tuesday
Monday
Saturday
Thursday
Sunday
Wednesday
```

Removing items from the set

Following methods used to remove the items from the set

1. **discard**
2. **remove**
3. **pop**

- discard() **method**

Python provides **discard()** method which can be used to remove the items from the set.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nRemoving some months from the set...");
Months.discard("January");
```

```
Months.discard("May");
print("\nPrinting the modified set...");
print(Months)
```

output:

```
{'February', 'January', 'March', 'April', 'June', 'May'}
Removing some months from the set...
Printing the modified set...
{'February', 'March', 'April', 'June'}
```

- remove() method

Remove "banana" by using the remove() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

output:

```
{"apple", "cherry"}
```

- pop() method

the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

Note: Sets are *unordered*, so when using the pop() method, you will not know which item that gets removed.

Example

Remove the last item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

output:

```
apple
{'cherry', 'banana'}
```

delete the set

The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

output:

```
File "demo_set_del.py", line 5, in <module>
  print(thisset) #this will raise an error because the set no longer exists
NameError: name 'thisset' is not defined
```

Difference between discard() and remove()

If the key to be deleted from the set using discard() doesn't exist in the set, the python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the python will give the error.

Adding items to the set

- add() method
- update() method.

Python provides the add() method which can be used to add some particular item to the set.

```
Months = set(["January","February", "March", "April", "May", "June"])
Months.add("July");
Months.add("August");
print(Months)
```

output:

```
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}
```

```
Months = set(["January","February", "March", "April", "May", "June"])
Months.update(["July","August","September","October"]);
print(Months)
```

output:

```
{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}
```

Python set operations (union, intersection, difference and symmetric difference)

In Python, below quick operands can be used for different operations.

```
| for union.
& for intersection.
```

– for difference
^ for symmetric difference

```
A = {0, 2, 4, 6, 8};
B = {1, 2, 3, 4, 5};

# union
print("Union :", A | B)

# intersection
print("Intersection :", A & B)

# difference
print("Difference :", A - B)

# symmetric difference
print("Symmetric difference :", A ^ B)
```

Output:

```
('Union :', set([0, 1, 2, 3, 4, 5, 6, 8]))

('Intersection :', set([2, 4]))

('Difference :', set([8, 0, 6]))

('Symmetric difference :', set([0, 1, 3, 5, 6, 8]))
```

Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with set to perform different tasks.

Function	Description
all()	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.

min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).
sum()	Retrun the sum of all elements in the set.

Dictionary

Dictionary is used to implement the key-value pair in python.

The keys are the immutable python object, i.e., Numbers, string or tuple.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:).

The collections of the key-value pairs are enclosed within the curly braces {}.

The syntax to define the dictionary is given below.

```
Dict = {"Name": "Ayush","Age": 22}
```

Accessing the dictionary values

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(Employee)
```

output:

```
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

The dictionary values can be accessed in the following way:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

print("printing Employee data .... ")
print("Name :",Employee["Name"])
print("Age : ",Employee["Age"])
print("Salary : ",Employee["salary"])
print("Company : ", Employee["Company"])
```

Updating dictionary values

Dictionary is mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dict = {'name':'MMP', 'age': 26}

# update value
my_dict['age'] = 27
print(my_dict)
```

Output: {'age': 27, 'name': 'MMP'}

```
# add item
my_dict['address'] = 'Downtown'
print(my_dict)
```

Output: {'address': 'Downtown', 'age': 27, 'name': 'MMP'}

Deleting elements using del keyword

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
```

Output:

```
printing the modified information
{'Age': 29, 'salary': 25000}
```

Dictionary Operations

Below is a list of common dictionary operations:

- create an empty dictionary

```
x = {}
```

- create a three items dictionary

```
x = {"one":1, "two":2, "three":3}
```

- access an element

```
x['two']
```

- get a list of all the keys

```
x.keys()
```

- get a list of all the values

```
x.values()
```

- add an entry
- x["four"]=4

- change an entry

```
x["one"] = "uno"
```

- delete an entry

```
del x["four"]
```

- remove all items

```
x.clear()
```

- number of items

```
z = len(x)
```

- looping over keys

for item in x.keys(): print item

- looping over values

for item in x.values(): print item

Built-in Dictionary functions

The built-in python dictionary methods along with the description are given below.

SN	Function	Description
1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.

Use of Python built-in functions(e.g. type/data conversion functions, math functions , etc)

1. int(a,base) : This function converts **any data type to integer**. 'Base' specifies the **base in which string is** if data type is string.

2. float() : This function is used to convert **any data type to a floating point number**

```
# Python code to demonstrate Type conversion
# initializing string
s = "10010"

# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ", end="")
print (c)

# printing string converting to float
e = float(s)
print ("After converting to float : ", end="")
print (e)
```

Output:

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

3. ord() : This function is used to convert a **character to integer.**

4. hex() : This function is to convert **integer to hexadecimal string**.

5. oct() : This function is to convert **integer to octal string**.

```
# initializing integer
s = '4'

# printing character converting to integer
c = ord(s)
print ("After converting character to integer : ",end="")
print (c)

# printing integer converting to hexadecimal string
c = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (c)

# printing integer converting to octal string
c = oct(56)
print ("After converting 56 to octal string : ",end="")
print (c)
```

Output:

```
After converting character to integer : 52

After converting 56 to hexadecimal string : 0x38

After converting 56 to octal string : 0o70
```

6. tuple() : This function is used to **convert to a tuple**.

7. set() : This function returns the **type after converting to set**.

8. list() : This function is used to convert **any data type to a list type**.

```
# Python code to demonstrate Type conversion
# using  tuple(), set(), list()

# initializing string
s = 'geeks'

# printing string converting to tuple
c = tuple(s)
print ("After converting string to tuple : ",end="")
print (c)

# printing string converting to set
c = set(s)
print ("After converting string to set : ",end="")
print (c)

# printing string converting to list
c = list(s)
print ("After converting string to list : ",end="")
print (c)
```

Output:

```
After converting string to tuple : ('g', 'e', 'e', 'k', 's')

After converting string to set : {'k', 'e', 's', 'g'}

After converting string to list : ['g', 'e', 'e', 'k', 's']
```

9. **dict()** : This function is used to **convert a tuple of order (key,value) into a dictionary**.
10**. str()** : Used to **convert integer into a string**.
11. **complex(real,imag)** : : This function **converts real numbers to complex(real,imag) number**.

```
# Python code to demonstrate Type conversion
# using  dict(), complex(), str()


# initializing integers
a = 1
b = 2

# initializing tuple
tup = (('a', 1) ,('f', 2), ('g', 3))

# printing integer converting to complex number
```

```
c = complex(1,2)
print ("After converting integer to complex number : ",end="")
print (c)

# printing integer converting to string
c = str(a)
print ("After converting integer to string : ",end="")
print (c)

# printing tuple converting to expression dictionary
c = dict(tup)
print ("After converting tuple to dictionary : ",end="")
print (c)
```

Output:

```
After converting integer to complex number : (1+2j)

After converting integer to string : 1

After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}
```

4.2 User defined functions: Function definition, function calling, function arguments and parameter passing, Return statement, Scope of Variables: Global variable and Local variable.

the user can create its functions which can be called user-defined functions.

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

```
def my_function():
        function code
        return <expression>
```

Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

```
def hello_world():
        print("hello world")
hello_world()
```

Output:

hello world

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses.

```
def hi(name):
   print(name)

hi("MMP")
```

Output:
MMP

```
def my_function(fname, lname):
        print(fname + " " + lname)

my_function("Purva","Pawar")
```

Output:
Purva Pawar

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[1])

my_function("purva","sandesh","jiyansh")
```

Output
The youngest child is sandesh

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
        print("Her last name is " + kid["lname"])

my_function(fname = "nitu", lname = "mini")
```

Output
Her last name is mini

Default Parameter Value

If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
    print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output
I am from Sweden
I am from India
I am from Norway
I am from Brazil

Passing a List as an Argument

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Output
apple
banana
cherry

Return statement

```
def my_function(x):

  return 5 * x


print(my_function(3))

print(my_function(5))
```

```
print(my_function(9))
```

Output

15

25

45

Scope of Variables: Global variable and Local variable.

Local variable

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.
A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Output

300

Global variable

Global variables are available from within any scope, global and local.
A variable created outside of a function is global and can be used by anyone:

```
x = 300

def myfunc():
        print(x)

myfunc()

print(x)
```

Output
300

300
The global keyword makes the variable global.

```
def myfunc():

        global x

         x = 300

myfunc()
```

```
print(x)
```

Output
300

Modules: Writing modules

Shown below is a Python script containing the definition of SayHello() function. It is saved as **hello.py**.

Example: hello.py

```
def SayHello(name):
   print("Hello {}! How are you?".format(name))
   return
```

importing modules

>>> import hello

>>> hello.SayHello("purva")

Output

Hello purva! How are you?

importing objects from modules

To import only parts from a module, by using the from keyword.

The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Output:

Python built-in modules(e.g. Numeric and Mathematical module, Functional programming module)

Python - Math Module

```
>>> import math
>>>math.pi
3.141592653589793

>>>math.log(10)
2.302585092994046

>>math.sin(0.5235987755982988)
0.49999999999999994
>>>math.cos(0.5235987755982988)
0.8660254037844387
>>>math.tan(0.5235987755982988)
0.5773502691896257

>>>math.radians(30)
0.5235987755982988
>>>math.degrees(math.pi/6)
29.999999999999996
```

Namespace and Scoping.

- A namespace is a mapping from names to objects.
- Python implements namespaces in the form of dictionaries.
- It maintains a name-to-object mapping where names act as keys and the objects as values.
- Multiple namespaces may have the same name but pointing to a different variable.

- A scope is a textual region of a Python program where a namespace is directly accessible.

  - Local scope
  - Non-local scope
  - Global scope
  - Built-ins scope

1. **The local scope.** The local scope is determined by whether you are in a class/function definition or not. Inside a class/function, the local scope refers to the names defined inside them. Outside a class/function, the local scope is the same as the global scope.

2. **The non-local scope.** A non-local scope is midways between the local scope and the global scope, e.g. the non-local scope of a function defined inside another function is the enclosing function itself.

3. **The global scope**. This refers to the scope outside any functions or class definitions. It also known as the module scope.

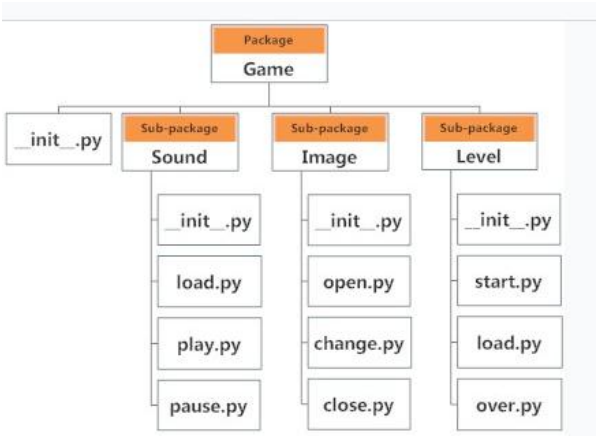
4. **The built-ins scope**. This scope, as the name suggests, is a scope that is built into Python. While it resides in its own module, any Python program is qualified to call the names defined here without requiring special access.

```
# var1 is in the global namespace
var1 = 5
def some_func():

    # var2 is in the local namespace
    var2 = 6
    def some_inner_func():

        # var3 is in the nested local
        # namespace
        var3 = 7
```

Python Packages : Introduction

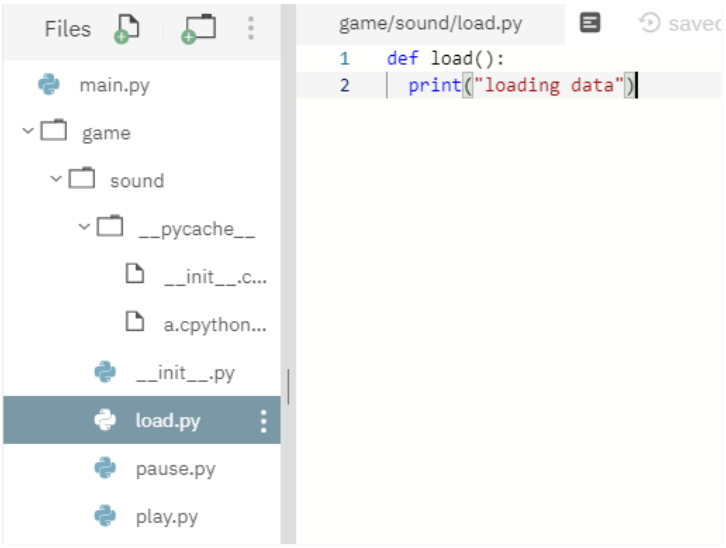
Python has packages for directories and modules for files. As a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

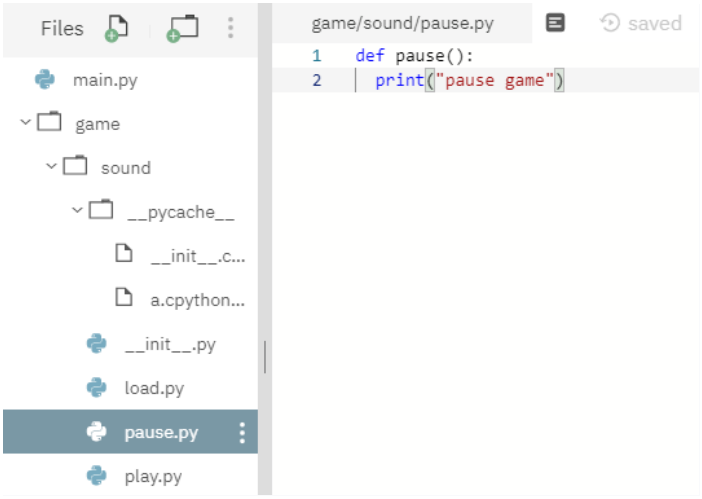
A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

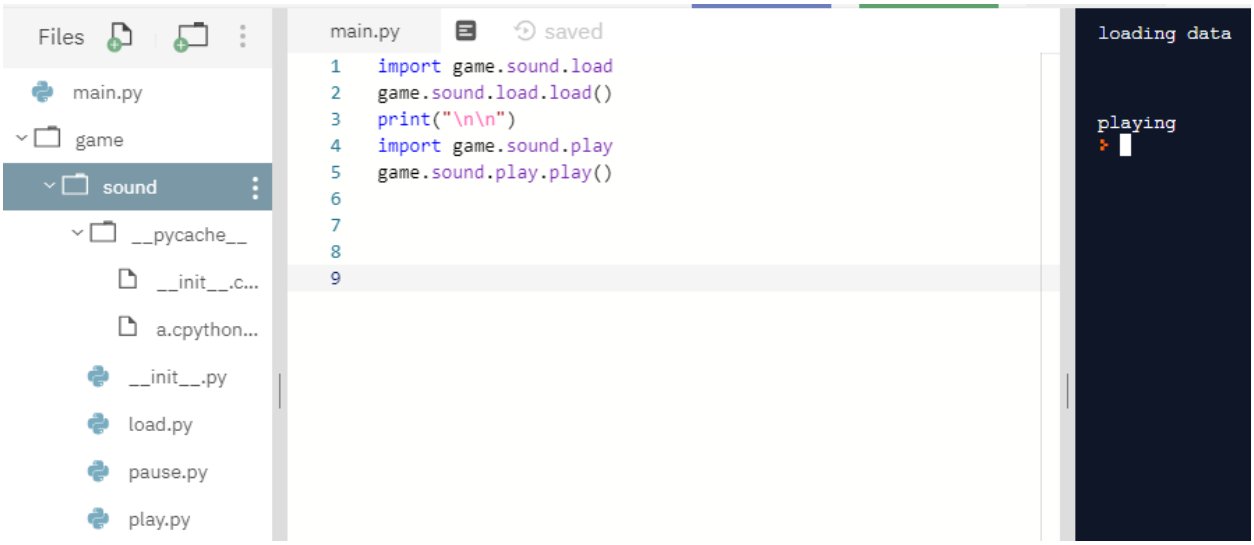


Steps:

- First create folder game.
- Inside it again create folder sound.
- Inside sound folder create load.py file.
- Inside sound folder create pause.py file.
- Inside sound folder create play.py file.
- Import package game and subpackage sound(files:load,pause,play)







Importing module from a package

import game.sound.load
Now if this module contains a function named load(), we must use the full name to reference it.

game.sound.load.load()

Math package:

>>> import math
>>>math.pi
3.141592653589793

sin, cos and tan ratios for the angle of 30 degrees (0.5235987755982988 radians):

>>math.sin(0.5235987755982988)

0.49999999999999994

>>>math.cos(0.5235987755982988)

0.8660254037844387

>>>math.tan(0.5235987755982988)

0.5773502691896257

NumPy is a python library used for working with arrays.

NumPy stands for Numerical Python.

Why Use NumPy ?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster that traditional Python lists.

- Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

Import NumPy

main.py saved

```
1  import numpy
2
3  arr = numpy.array([1, 2, 3, 4, 5])
4
5  print(arr)
```

```
[1 2 3 4 5]
```

Use a tuple to create a NumPy array:

main.py saving...

```
1  import numpy as np
2
3  arr = np.array((1, 2, 3, 4, 5))
4
5  print(arr)
6
```

```
[1 2 3 4 5]
```

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Create a 0-D array with value 42

main.py saving...

```
1  import numpy as np
2
3  arr = np.array(42)
4
5  print(arr)
```

```
42
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

Create a 1-D array containing the values 1,2,3,4,5:

main.py saving...

```
1  import numpy as np
2
3  arr = np.array([1, 2, 3, 4, 5])
4
5  print(arr)
```

```
[1 2 3 4 5]
```

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

main.py saving...

```
1  import numpy as np
2
3  arr = np.array([[1, 2, 3], [4, 5, 6]])
4
5  print(arr)
```

```
[[1 2 3]
 [4 5 6]]
```

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

main.py saved

```
1  import numpy as np
2
3  arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]
   ]])
4
5  print(arr)
```

```
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

SciPy package

SciPy, pronounced as Sigh Pi, is a scientific python open source, distributed under the BSD licensed library to perform Mathematical, Scientific and Engineering Computations..

The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation.

SciPy Sub-packages

- SciPy consists of all the numerical code.

- SciPy is organized into sub-packages covering different scientific computing domains. These are summarized in the following table −

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions

scipy.stats	Statistics

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.
It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+. ... SciPy makes use of Matplotlib.

Pandas is used for data manipulation, analysis and cleaning. Python pandas is well suited for different kinds of data, such as:

- Tabular data with heterogeneously-typed columns
- Ordered and unordered time series data
- Arbitrary matrix data with row & column labels
- Any other form of observational or statistical data sets

Chapter 5.

Object Oriented Programming in Python

12 Marks

Introduction:

- Python follows ***object oriented programming paradigm***. It deals with declaring python classes and objects which lays the foundation of OOP's concepts.
- Python programming offers OOP style programming and provides an easy way to develop programs. It uses the OOP concepts that makes python ***more powerful to help design a program that represents real world entities***.
- Python supports OOP concepts such as ***Inheritance, Method Overriding, Data abstraction and Data hiding***.

Important terms in OOP/ Terminology of OOP-

(Different OOP features supported by Python)

- **Class**- Classes are defined by the user. The class ***provides the basic structure for an object***. It ***consists of data members and method members*** that are used by the instances(object) of the class.
- **Object**- A unique ***instance of a data structure*** that is defined by its class. An object ***comprises both data members and methods***. Class itself does nothing but real functionality is achieved through their objects.

- **Data Member**: A variable ***defined in either a class or an object***; it holds the data associated with the class or object.
- **Instance variable**: A variable that is ***defined in a method***, its scope is only within the object that defines it.
- **Class variable**: A variable that is ***defined in the class*** and can be used by all the instance of that class.
- **Instance:** An object is an instance of a class.
- **Method**: They are ***functions that are defined in the definition of class*** and are used by various instances of the class.
- **Function Overloading**: A function ***defined more than one time*** with different behavior. (different arguments)
- **Encapsulation**: It is the ***process of binding together the methods and data variables as a single entity*** i.e. class. It hides the data within the class and makes it available only through the methods.
- **Inheritance**: The ***transfer of characteristics of a class to other classes*** that are derived from it.
- **Polymorphism**: It allows ***one interface to be used for a set of actions***. It means same function name(but different signatures) being used for different types.
- **Data abstraction**: It is the ***process of hiding the implementation details and showing only functionality*** to the user.

Classes-

- Python is OOP language. Almost everything in python is an object with its properties and methods.
- Object is simply a collection of data(variables) and methods(functions) that acts on those data.

Creating Classes:

A class is a block of statement that combine data and operations,

which are performed on the data, into a group as a single unit and

acts as a blueprint for the creation of objects.

Syntax:

class ClassName:

‘ Optional class documentation string

#list of python class variables

Python class constructor

#Python class method definitions

- In a class we can define variables, functions etc. While writing ***function in class we have to pass atleast one argument*** that is called ***self parameter***.
- The self parameter is a reference to the class itself and is used to access variables that belongs to the class.

Example: Creating class in .py file

```
class student:
    def display(self):  # defining method in class
        print("Hello Python")
```

- In python programming **self is a default variable** that contains the ***memory address of the instance of the current class.***
- So we can use self to refer to all the instance variable and instance methods.

Objects and Creating Objects-

- An object is an instance of a class that has some attributes and behavior.
- Objects can be used to access the attributes of the class.

Example:

```
class student:
    def display(self):  # defining method in class
        print("Hello Python")
s1=student()     #creating object of class
s1.display()     #calling method of class using object
```

Output:

```
Hello Python
```

Example: Class with get and put method

```
class car:
    def get(self,color,style):
        self.color=color
        self.style=style
    def put(self):
        print(self.color)
        print(self.style)
c=car()
c.get('Brio','Red')
c.put()
```

Output*:*

```
Brio
Red
```

Instance variable and Class variable:

- **Instance variable** is ***defined in a method and its scope is only within the object*** that defines it.
- Every object of the class has its own copy of that variable. Any changes made to the variable don’t reflect in other objects of that class.

- **Class variable** is ***defined in the class*** and can be ***used by all the instances of that class***.
- ***Instance variables are unique for each instance***, while ***class variables are shared by all instances***.

Example: For instance and class variables

```
class sample:
    x=2      # x is class variable
    def get(self,y):  # y is instance variable
        self.y=y
s1=sample()
s1.get(3)   # Access attributes
print(s1.x," ",s1.y)
s2=sample()
s2.y=4
print(s2.x," ",s2.y)
```

Output:

```
2   3
2   4
```

Data Hiding:

- Data hiding is ***a software development technique specifically used in object oriented programming to hide internal object details***(data members).
- It ***ensures exclusive data access to class members*** and ***protects object integrity by preventing unintended or intended changes***.
- Data hiding is also known as **information hiding**. An objects attributes may or may not be visible outside the class definition.
- We need to **name attributes with a double underscore**(_ _) **prefix** and ***those attributes the are not directly visible to outsiders***. Any variable prefix with double underscore is called **private variable** which is accessible only with class where it is declared.

Example: For data hiding

```
class counter:
    __secretcount=0   # private variable
    def count(self):  # public method
        self.__secretcount+=1
        print("count= ",self.__secretcount)  # accessible in the same class
c1=counter()
c1.count() # invoke method
c1.count()
print("Total count= ",c1.__secretcount) # cannot access private variable directly
```

Output:

```
count=  1
count=  2
```

Traceback (most recent call last):

File "D:\python programs\class_method.py", line 9, in <module>

print("Total count= ",c1.__secretcount) # cannot access private variable directly

AttributeError: 'counter' object has no attribute '__secretcount'

Data Encapsulation and Data Abstraction:

- We can ***restrict access of methods and variables in a class with the help of encapsulation***. It will prevent the data being modified by accident.
- **Encapsulation *is used to hide the value or state of a structured data object inside a class***, preventing unauthorized parties direct access to them.
- **Data abstraction** refers to ***providing only essential information about the data to the outside world, hiding the background details of implementation***.
- **Encapsulation** is a ***process to bind data and functions together into a single unit i.e. class*** while **abstraction** is a process in which the ***data inside the class is the hidden from outside world***.
- In short hiding internal details and showing functionality is known as **abstraction**.
- To support encapsulation, declare the methods or variables as private in the class. The ***private methods cannot be called by the object directly***. It can be called only from within the class in which they are defined.
- Any ***function with double underscore is called private method***.

Access modifiers for variables and methods are:

- **Public methods / variables-** Accessible from anywhere inside the class, in the sub class, in same script file as well as outside the script file.

- **Private methods / variables**- Accessible only in their own class. Starts with two underscores.

Example: For access modifiers with data abstraction

```
class student:
    __a=10 #private variable
    b=20 #public variable
    def __private_method(self): #private method
        print("Private method is called")
    def public_method(self): #public method
        print("public method is called")
        print("a= ",self.__a) #can be accessible in same class
s1=student()
# print("a= ",s1.__a) #generate error
print("b=",s1.b)
# s1.__private_method() #generate error
s1.public_method()
```

Output:

```
b= 20
public method is called
a=  10
```

Creating Constructor:

- Constructors are generally used for ***instantiating an object***.
- The task of constructors is to ***initialize***(assign values) to the data members of the class when an object of class is created.
- In Python the **__init__()** method is called the constructor and is always called when an object is created.

***Syntax of constructor declaration* :**

```
def __init__(self):
    # body of the constructor
```

Example: ***For creating constructor use_ _init_ _ method called as constructor***.

```
class student:
    def __init__(self,rollno,name,age):
        self.rollno=rollno
        self.name=name
        self.age=age
        print("student object is created")
p1=student(11,"Ajay",20)
print("Roll No of student= ",p1.rollno)
print("Name No of student= ",p1.name)
print("Age No of student= ",p1.age)
```

***Output**:*

student object is created

Roll No of student= 11

Name No of student= Ajay

Age No of student= 20

Programs:

Define a class rectangle using length and width.It has a method which can compute area.

```
class rectangle:
    def __init__(self,L,W):
        self.L=L
        self.W=W
    def area(self):
        return self.L*self.W
r=rectangle(2,10)
print(r.area())
```

Output

20

Create a circle class and initialize it with radius. Make two methods getarea and getcircumference inside this class

```
class circle:
    def __init__(self,radius):
        self.radius=radius
    def getarea(self):
        return 3.14*self.radius*self.radius
    def getcircumference(self):
        return 2*3.14*self.radius
c=circle(5)
print("Area=",c.getarea())
print("Circumference=",c.getcircumference())
```

Output:

```
Area= 78.5
Circumference= 31.400000000000002
```

Types of Constructor:

There are two types of constructor- ***Default constructor and Parameterized constructor***.

Default constructor- The default constructor is simple constructor which does not accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

Example: **Display Hello message using Default constructor(It does not accept argument)**

```
class student:
    def __init__(self):
        print("This is non parameterized constructor")
    def show(self,name):
        print("Hello",name)
s1=student()
s1.show("World")
```

Output:

```
This is non parameterized constructor
Hello World
```

Example: **Counting the number of objects of a class**

```
class student:
    count=0
    def __init__(self):
        student.count=student.count+1
```

```
s1=student()

s2=student()

print("The number of student objects",student.count)
```

Output:

```
The number of student objects 2
```

Parameterized constructor- Constructor with parameters is known as parameterized constructor.

The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example: For parameterized constructor

```
class student:
    def __init__(self,name):
        print("This is parameterized constructor")
        self.name=name
    def show(self):
        print("Hello",self.name)
s1=student("World")
```

s1.show()

Output:

This is parameterized constructor

Hello World

Destructor:

A class can define a special method called destructor with the help of **_ _del_ _()**. It is invoked automatically when the instance (object) is about to be destroyed.

It is mostly ***used to clean up non memory resources*** used by an instance(object).

Example: For Destructor

```
class student:
    def __init__(self):
        print("This is non parameterized constructor")

    def __del__(self):
        print("Destructor called")
s1=student()
s2=student()
del s1
```

Output:

This is non parameterized constructor

This is non parameterized constructor

Destructor called

Method Overloading:

- ***Method overloading is the ability to define the method with the same name but with a different number of arguments and data types***.
- With this ability one method can perform different tasks, depending on the number of arguments or the types of the arguments given.
- Method overloading is a concept in which a method in a class performs operations according to the parameters passed to it.
- As in other language we can write a program having two methods with same name but with different number of arguments or order of arguments but in python if we will try to do the same we get the following issue with method overloading in python.

  Example-

  # To calculate area of rectangle

  ```
  def area(length,breadth):
      calc=length*breadth
      print(calc)
  # To calculate area of square
  def area(size):
      calc=size*size
  ```

```
    print(calc)
area(3)
area(4,5)
```

Output-

```
9
Traceback (most recent call last):
  File "D:\python programs\trial.py", line 10, in <module>
    area(4,5)
TypeError: area() takes 1 positional argument but 2 were given
```

- Python does not support method overloading i.e it is not possible to define more than one method with the same name in a class in python.
- This is because method arguments in python do not have a type. A method accepting one argument can be called with an integer value, a string or a double as shown in example.

Example-

```
class demo:
    def print_r(self,a,b):
        print(a)
        print(b)
obj=demo()
obj.print_r(10,'S')
```

```
obj.print_r('S',10)
```

Output:

```
10
S
S
10
```

- In the above example same method works for two different data types.
- It is clear that method overloading is not supported in python but that does not mean that we cannot call a method with different number of arguments. There are couple of alternatives available in python that make it possible to call the same method but with different number of arguments.

Using Default Arguments:

It is possible to provide default values to method arguments while defining a method. If method arguments are supplied default values, then it is not mandatory to supply those arguments while calling method as shown in example.

***Example 1*: Method overloading with deafult arguments**

```
class demo:
    def arguments(self,a=None,b=None,c=None):
        if(a!=None and b!=None and c!=None):
            print("3 arguments")
        elif (a!=None and b!=None):
```

```
            print("2 arguments")
        elif a!=None:
            print("1 argument")
        else:
            print("0 arguments")
obj=demo()
obj.arguments("Amol","Kedar","Sanjay")
obj.arguments("Amit","Rahul")
obj.arguments("Sidharth")
obj.arguments()
```

Output-

```
3 arguments
2 arguments
1 argument
0 arguments
```

***Example 2*: With a method to perform different operations using method overloading**

```
class operation:
    def add(self,a,b):
        return a+b
op=operation()
# To add two integer numbers
print("Addition of integer numbers= ",op.add(10,20))
# To add two floating numbers
print("Addition of integer numbers= ",op.add(11.12,12.13))
# To add two strings
print("Addition of stings= ",op.add("Hello","World"))
```

Output-

```
Addition of integer numbers=  30
Addition of integer numbers=  23.25
Addition of stings=  HelloWorld
```

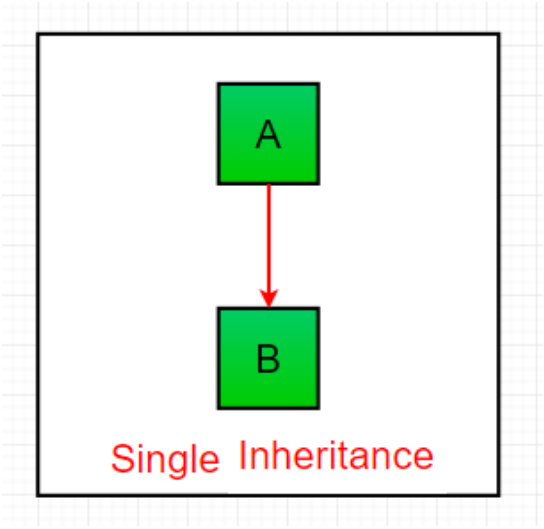
Inheritance:

The mechanism of designing and constructing classes from other classes is called inheritance.

Inheritance is the capability of one class to derive or inherit the properties from some another class.

The new class is called **derived class** or **child class** and the class from which this derived class has been inherited is the **base class** or **parent class**. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.



Syntax:

```
Class A:
        # Properties of class A
Class B(A):
        # Class B inheriting property of class A
        #  more properties of class B
```

Example 1: **Example of Inheritance without using constructor**

```
class vehicle:  #parent class
    name="Maruti"
    def display(self):
        print("Name= ",self.name)
class category(vehicle): # drived class
    price=400000
    def disp_price(self):
        print("price= ",self.price)
car1=category()
car1.display()
car1.disp_price()
```

Output:

Name= Maruti

price= 400000

Example 2: **Example of Inheritance using constructor**

```
class vehicle:  #parent class
    def __init__(self,name,price):
        self.name=name
        self.price=price
    def display(self):
        print("Name= ",self.name)
class category(vehicle): # drived class
    def __init__(self,name,price):
        vehicle.__init__(self,name,price) #pass data to base constructor
    def disp_price(self):
        print("price= ",self.price)
car1=category("Maruti",400000)
car1.display()
car1.disp_price()
```

car2=category("Honda",600000)

car2.display()

car2.disp_price()

Output:

Name= Maruti

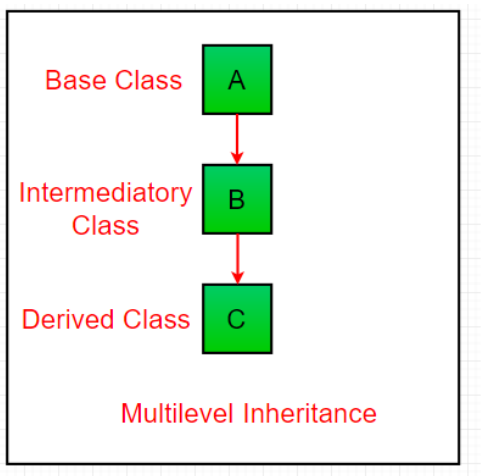
price= 400000

Name= Honda

price= 600000

Multilevel Inheritance:

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



Syntax:

```
Class A:
        # Properties of class A
Class B(A):
        # Class B inheriting property of class A
        #  more properties of class B
Class C(B):
        # Class C inheriting property of class B
        # thus, Class C also inherits properties of class A
        # more properties of class C
```

Example 1: Python program to demonstrate multilevel inheritance

```
#Mutilevel Inheritance
class c1:
    def display1(self):
        print("class c1")
class c2(c1):
    def display2(self):
        print("class c2")
class c3(c2):
```

```
    def display3(self):
        print("class c3")
s1=c3()
s1.display3()
s1.display2()
s1.display1()
```

Output:

```
class c3
class c2
class c1
```

Example 2: Python program to demonstrate multilevel inheritance

```
# Base class
class Grandfather:
    grandfathername =""
    def grandfather(self):
        print(self.grandfathername)
```

```
# Intermediate class
class Father(Grandfather):
    fathername = ""
    def father(self):
        print(self.fathername)

# Derived class
class Son(Father):
    def parent(self):
        print("GrandFather :", self.grandfathername)
        print("Father :", self.fathername)

# Driver's code
s1 = Son()
s1.grandfathername = "Srinivas"
s1.fathername = "Ankush"
s1.parent()
```

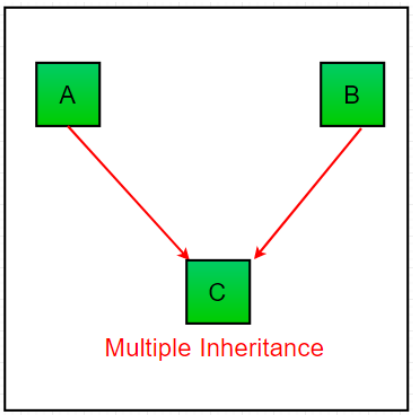
Output:

```
GrandFather : Srinivas
```

Father : Ankush

Multiple Inheritance:

When a class can be derived from more than one base classes this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.



Syntax:

Class A:

    # variable of class A

    # functions of class A

Class B:

    # variable of class B

    # functions of class B

Class C(A,B):

```
        # Class C inheriting property of both class A and B
        # more properties of class C
```

Example: **Python program to demonstrate multiple inheritance**

```
# Base class1
class Father:
    def display1(self):
        print("Father")
# Base class2
class Mother:
    def display2(self):
        print("Mother")
# Derived class
class Son(Father,Mother):
    def display3(self):
        print("Son")
s1 = Son()
s1.display3()
s1.display2()
s1.display1()
```

Output:

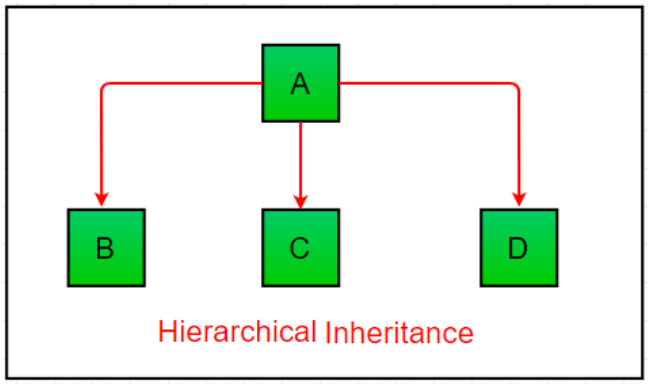
Son

Mother

Father

Hierarchical Inheritance:

When more than one derived classes are created from a single base this type of inheritence is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



***Example* : Python program to demonstrate Hierarchical inheritance**

```
# Base class

class Parent:
```

```
    def func1(self):
        print("This function is in parent class.")


# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")


# Derived class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")


object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

This function is in parent class.

This function is in child 1.

This function is in parent class.

This function is in child 2.

Method Overriding:

Method overriding is an ability of a class to change the implementation of a method provided by one of its base class. Method overriding is thus a strict part of inheritance mechanism.

To override a method in base class, we must define a new method with sam name and same parameters in the derived class.

Overriding is a very important part of OOP since it is feature that makes inheritance exploit its full power. Through method overriding a class may “copy” another class, avoiding duplicated code and at the same time enhance or customize part of it.

Example: For method overriding

```
class A:
    def display(self):
        print("This is base class")
```

```
class B(A):
    def display(self):
        print("This is derived class")
obj=B()              # instance of child
obj.display()        # child class overriden method
```

Output-

```
This is derived class
```

Using super() Method:

The super() method gives you access to methods in a super class from the subclass that inherits from it.

The super() method returns a temporary object of the superclass that then allows you to call that superclass's method.

Example: For method overriding with super()

```
class A:
    def display(self):
        print("This is base class")
class B(A):
    def display(self):
        super().display()
```

```
    print("This is derived class")
obj=B()              # instance of child
obj.display()        # child class overriden method
```

Output-

```
This is base class
This is derived class
```

Composition Classes:

- In composition we ***do not inherit from the base class*** but ***establish relationship between classes through the use of instance variables*** that are references to other objects.
- Composition means that ***an object knows another object and explicitly delegates some tasks to it***. While ***inheritance is implicit***, ***composition is explicit*** in python.
- We use composition when we want ***to use some aspects of another class without promising all of the features of that other class***.

Syntax:

```
Class GenericClass:
        Define some attributes and methods
```

```
Class AspecificClass:
        Instance_variable_of_generic_class=GenericClass
        #use this instance somewhere in the class
        Some_method(instance_varable_of_generic_class)
```

- For example, we have three classes email, gmail and yahoo. In email class we are referring the gmail and using the concept of composition.

Example:

```
class gmail:
    def send_email(self,msg):
        print("sending '{}' from gmail".format(msg))
class yahoo:
    def send_email(self,msg):
        print("sending '{}' from yahoo".format(msg))
class email:
    provider=gmail()
    def set_provider(self,provider):
        self.provider=provider
    def send_email(self,msg):
```

```
        self.provider.send_email(msg)
client1=email()
client1.send_email("Hello")
client1.set_provider(yahoo())
client1.send_email("Hello")
```

Output:

```
sending 'Hello' from gmail
sending 'Hello' from yahoo
```

Customization via Inheritance specializing inherited methods:

- The ***tree-searching model of inheritance*** turns out to be a great way to specialize systems. Because ***inheritance finds names in subclasses before it checks superclasses***, subclasses can replace default behavior by redefining the superclass's attributes.
- In fact, you can build entire systems as hierarchies of classes, which are extended by adding new external subclasses rather than changing existing logic in place.
- ***The idea of redefining inherited names leads to a variety of specialization techniques***.

- For instance, ***subclasses may replace inherited attributes completely, provide attributes that a superclass expects to find, and extend superclass methods by calling back to the superclass from an overridden method***.

***Example*- For specilaized inherited methods**

```
class A:
    "parent class"  #parent class
    def display(self):
        print("This is base class")
class B(A):
    "Child class"    #derived class
    def display(self):
        A.display(self)
        print("This is derived class")
obj=B()   #instance of child
obj.display()   #child calls overridden method
```

Output:

```
This is base class
This is derived class
```

- In the above example derived class.display() just extends base class.display() behavior rather than replacing it completely.
- ***Extension is the only way to interface with a superclass***.
- The following program defines multiple classes that illustrate a variety of common techniques.

Super

Defines a method function and a delegate that expects an action in a

subclass

Inheritor

Doesn't provide any new names, so it gets everything defined in Super

Replacer

Overrides Super's method with a version of its own

Extender

Customizes Super's method by overriding and calling back to run the

default

Provider

Implements the action method expected by Super's delegate method

***Example*- Various ways to customize a common superclass**

```
class super:
    def method(self):
        print("in super.method")  #default behavior
    def delegate(self):
        self.action()     #expected to be defined
class inheritor(super):
    pass
class replacer(super):  #replace method completely
    def method(self):
        print("in replacer.method")
class extender(super):    #extend method behavior
    def method(self):
        super.method(self)
        print("in extender.method")
class provider(super):   # fill in a required method
    def action(self):
        print("in provider.action“)
for klass in (inheritor,replacer,extender):
```

```
    print("\n"+klass.__name__+"...")
    klass().method()
    print("\n provider...")
x=provider()
x.delegate()
```

Output:

```
inheritor...
in super.method
 provider...
replacer...
in replacer.method
 provider...
extender...
in super.method
in extender.method
 provider...
in provider.action
```

- When we call the delegate method through provider instance, two independent inheritance searches occur:
- On the initial x.delegate call, Python finds the delegate method in Super, by searching at the provider instance and above. The instance x is passed into the method's self argument as usual.
- Inside the super.delegate method, self.action invokes a new, independent inheritance search at self and above. Because self references a provider instance, the action method is located in the provider subclass.

6.1 I/O Operations: Reading keyboard input , Printing to screen

Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are –

- raw_input
- input

The *raw_input* Function

The raw_input([prompt]) function reads one line from standard input and returns it as a string.

```
>>> str = raw_input("Enter your input: ")
Enter your input: mmpolytechnic
>>> print(str)
mmpolytechnic
>>>
```

The input Function

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
>>> str = input("Enter your name: ")
Enter your name: purva
Error:
NameError: name 'purva' is not defined(“for string input” instead of input use raw_input for accepting string value from user)
>>> str = input("Enter your input: ")
Enter your input: 1
>>> print(str)
```

"6.2 File Handling: Opening file in different modes, accessing file contents using standard library functions, Reading and writing files, closing a file, Renaming and deleting file, Directories in python, File and related standard functions

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

Opening file in different modes

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

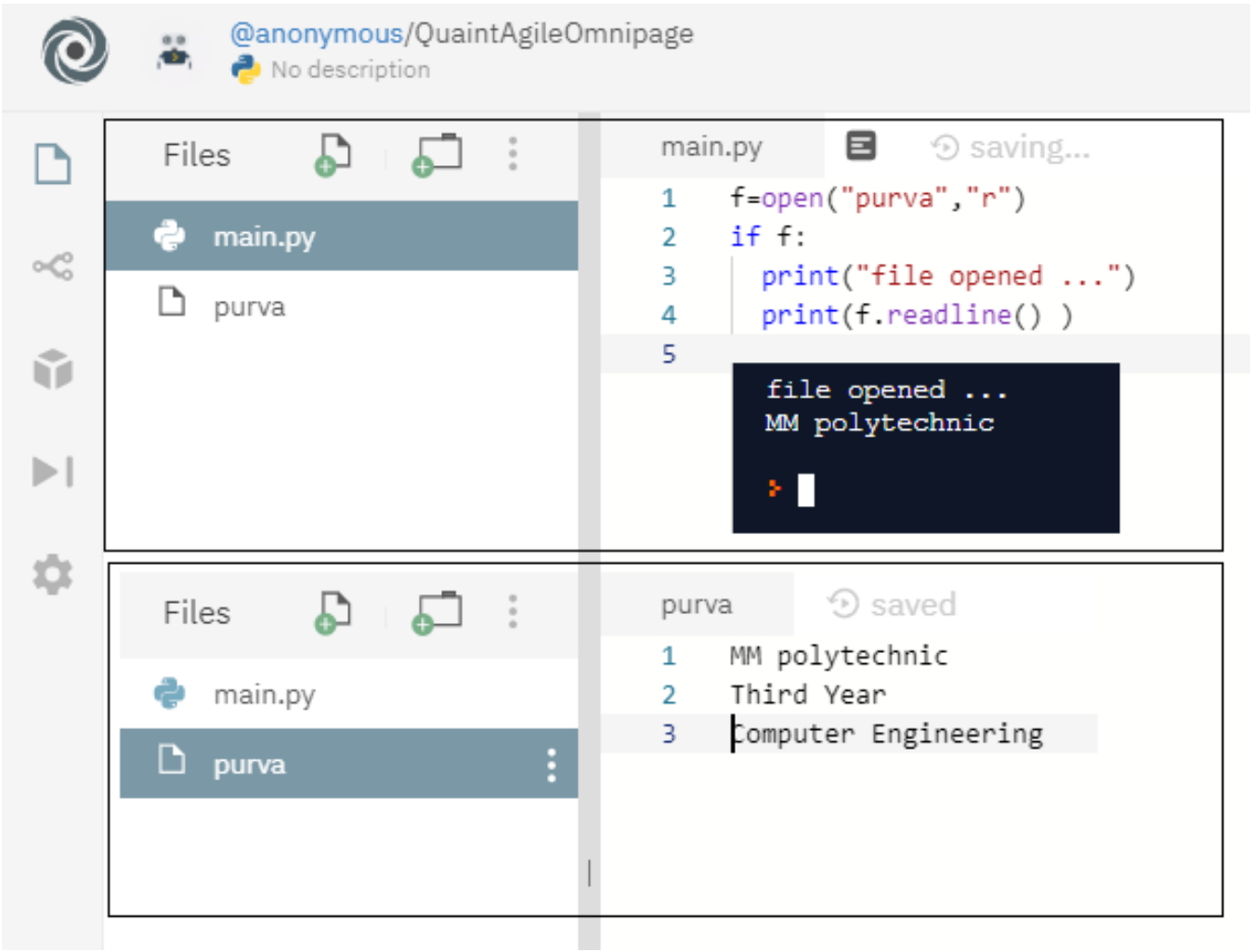
"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

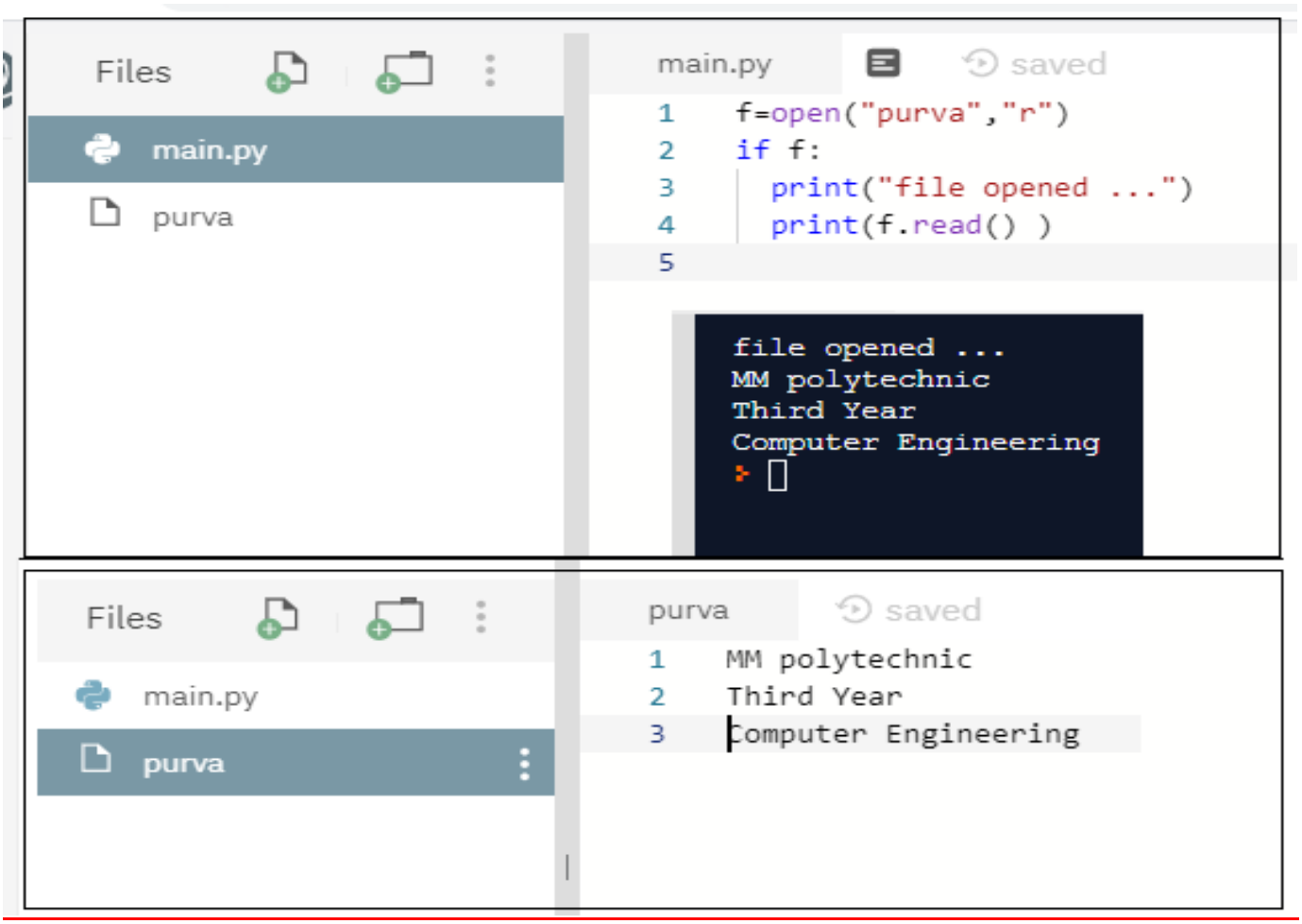
"b" - Binary - Binary mode (e.g. images)



Accessing file contents using standard library functions

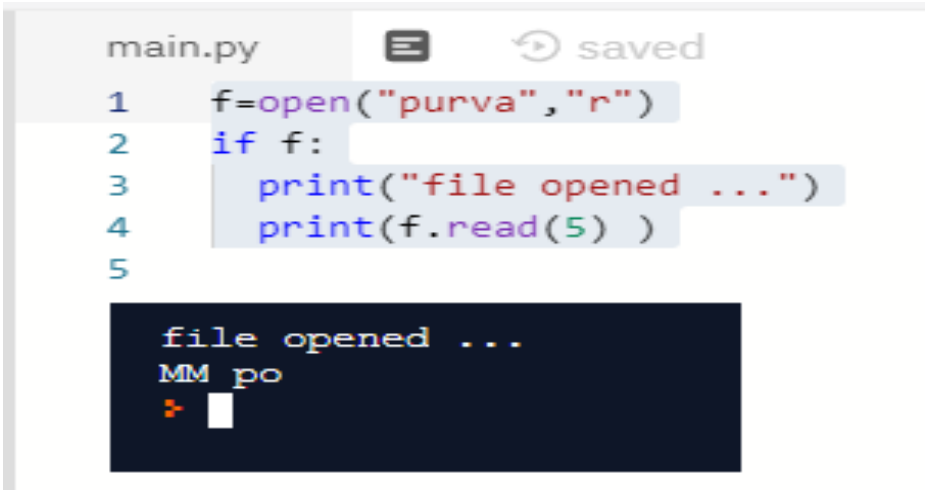
Open a File on the Server

Assume we have the following file, located in the same folder as Python:



Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:



Readline

You can return one line by using the readline() method:

main.py saved

```
1  f=open("purva","r")
2  if f:
3      print("file opened ...")
4      print(f.readline() )
5
```

```
file opened ...
MM polytechnic

>
```

Calling readline() 2 times

main.py saving...

```
1  f=open("purva","r")
2  if f:
3      print("file opened ...")
4      print(f.readline() )
5      print(f.readline() )
```

```
file opened ...
MM polytechnic

Third Year

>
```

By looping through the lines of the file, read the whole file, line by line:

main.py saving...

```
1   f=open("purva","r")
2   if f:
3     print("file opened ...")
4
5   for a in f:
6     print(a)
7
```

```
file opened ...
MM polytechnic

Third Year

Computer Engineering
>
```

Readlines

Read and return a list of lines from the file. Reads in at most n bytes/ characters if specified.

main.py saved

```
1   f=open("purva","r")
2   print("file opened ...")
3   print(f.readlines())
4
```

```
file opened ...
['MM polytechnic\n', 'Third Year\n', 'Computer Engineering']
>
```

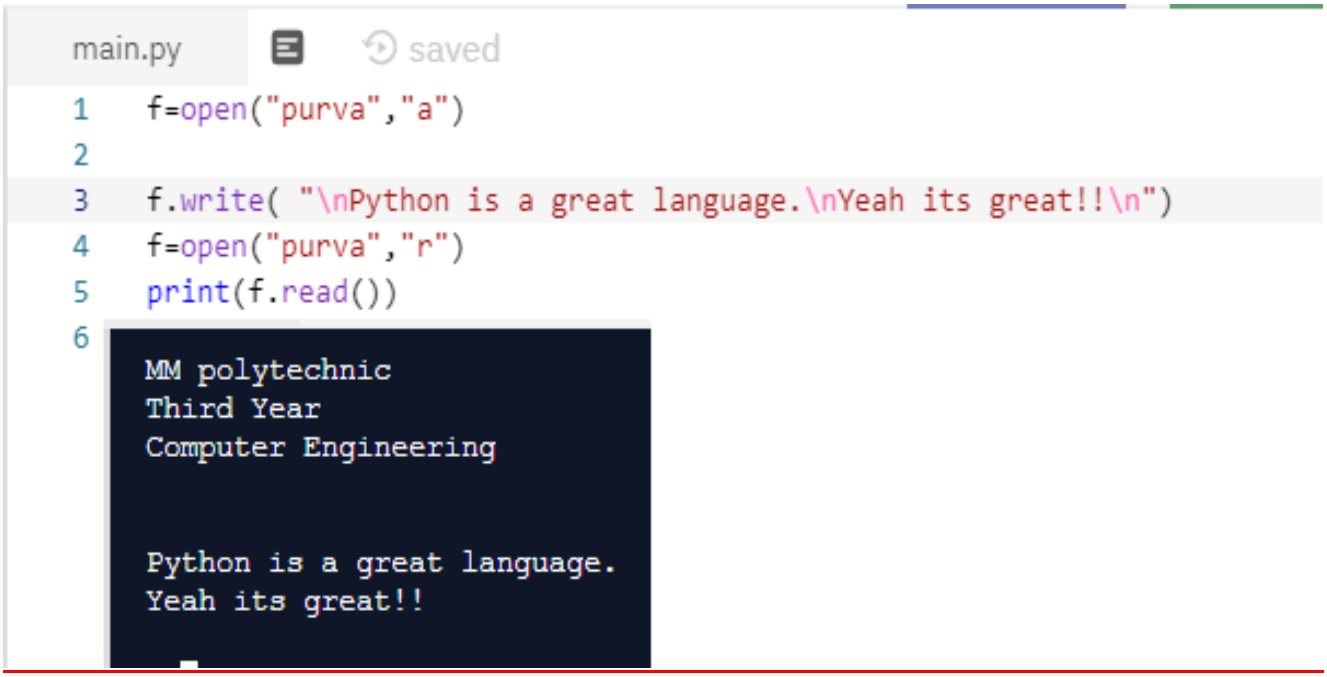
Reading and writing files

The write() Method

The write() method writes any string to an open file.

Python strings can have binary data and not just text.

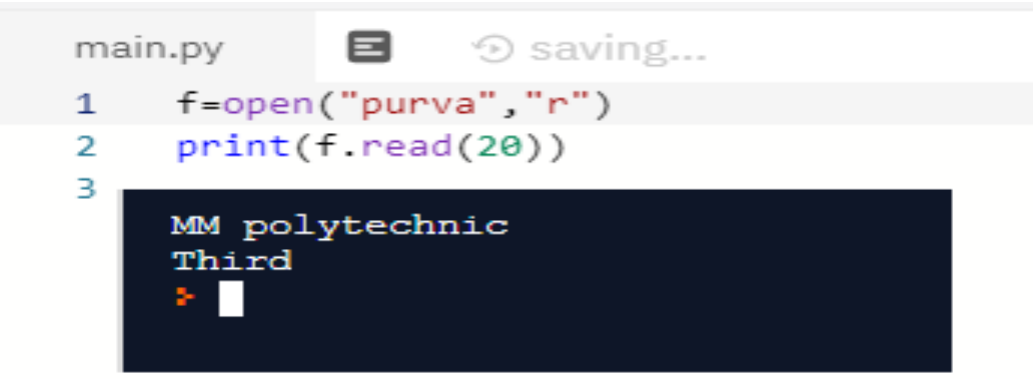
The write() method does not add a newline character ('\n') to the end of the string –



The read() Method

The read() method reads a string from an open file.

This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.



Closing a file

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

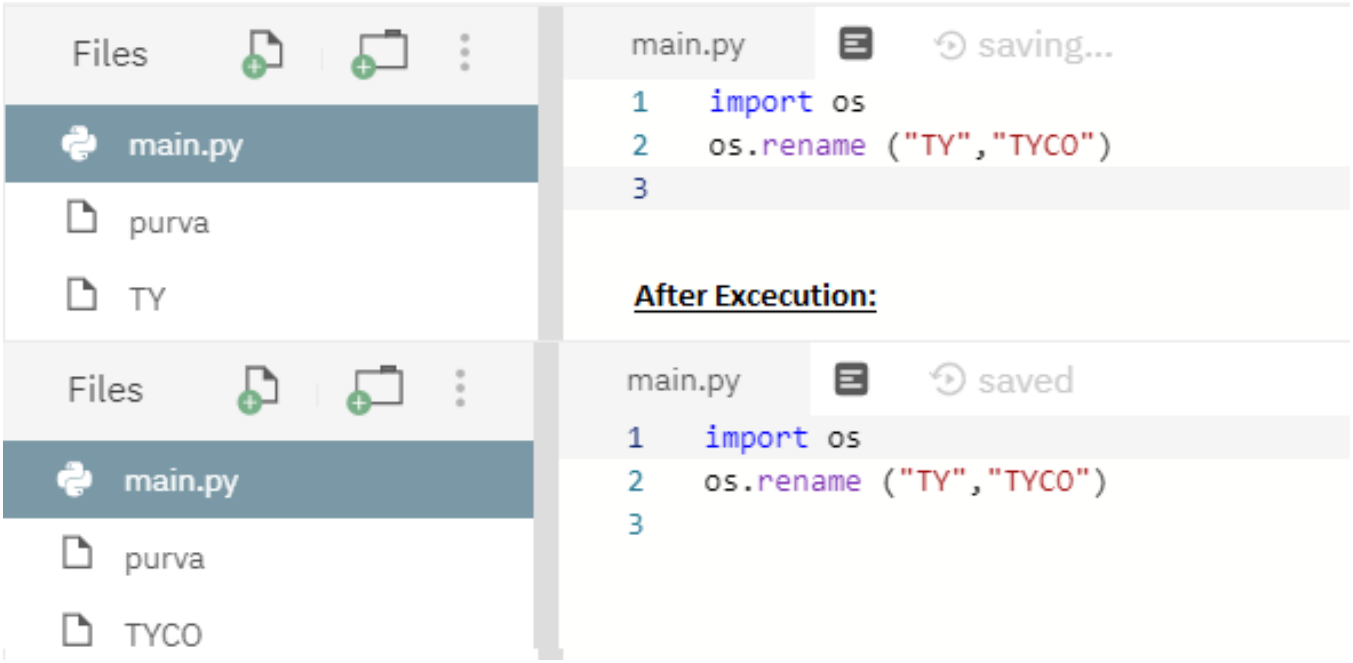
Renaming and deleting file

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then call any related functions.

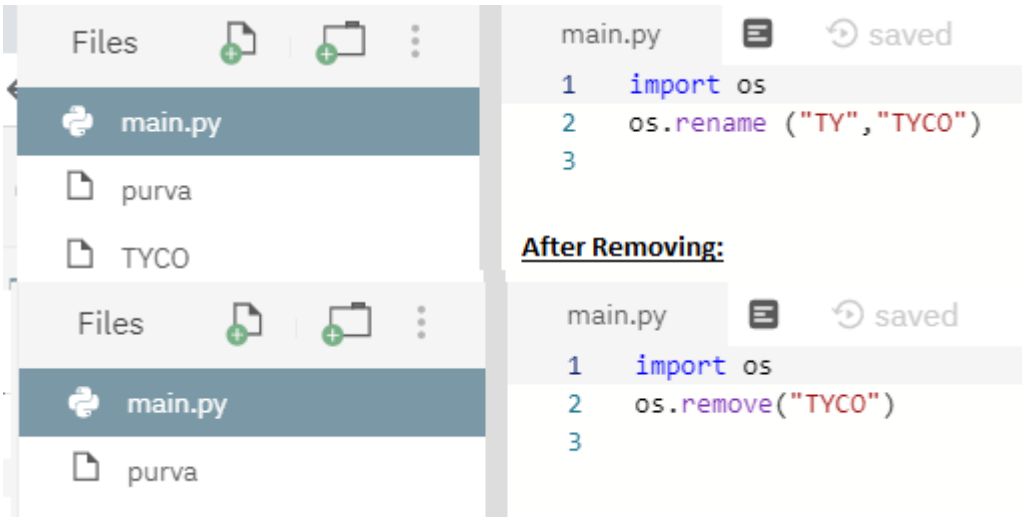
The rename() Method

The rename() method takes two arguments, the current filename and the new filename.



The remove() Method

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.



"6.3 Exception Handling:

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after that.

Common Exceptions

A list of common exceptions that can be thrown from a normal python program is given below.

ZeroDivisionError: Occurs when a number is divided by zero.

NameError: It occurs when a name is not found. It may be local or global.

IndentationError: If incorrect indentation is given.

IOError: It occurs when Input Output operation fails.

EOFError: It occurs when the end of the file is reached, and yet operations are being performed.

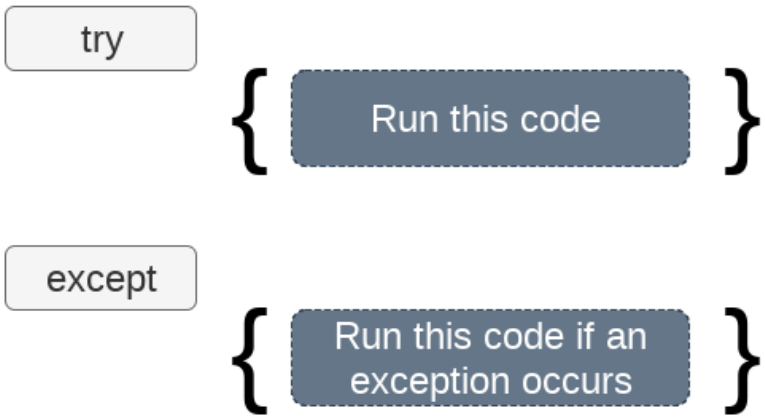
Exception Handling- ‘ try: except:’statement

- The **try** block lets you test a block of code for errors.
- The **except** block lets you handle the error.
- The **finally** block lets you execute code, regardless of the result of the try- and except blocks.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

Using multiple exceptions:



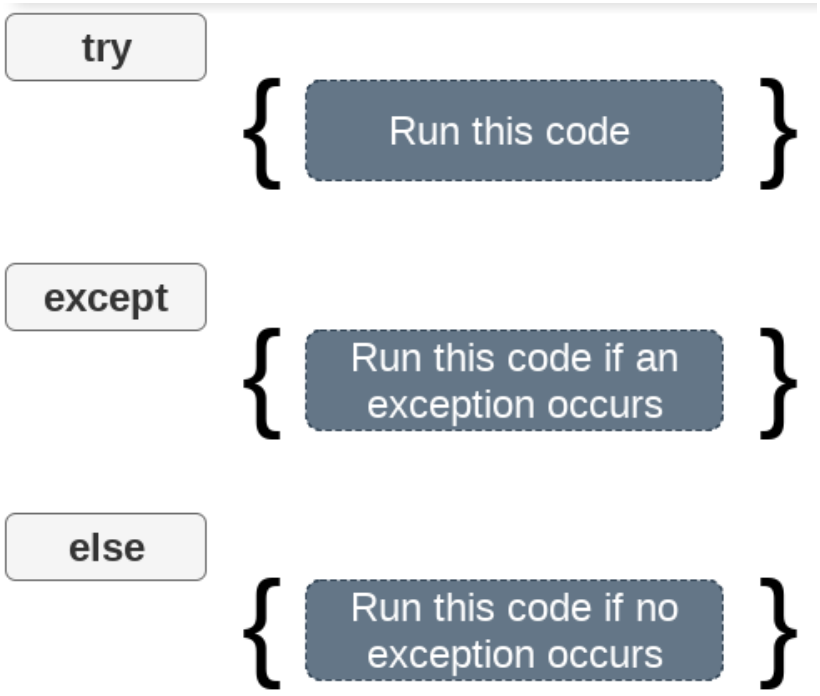

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

```
Variable x is not defined
```

The use the else statement with the try-except statement, place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.



main.py saved

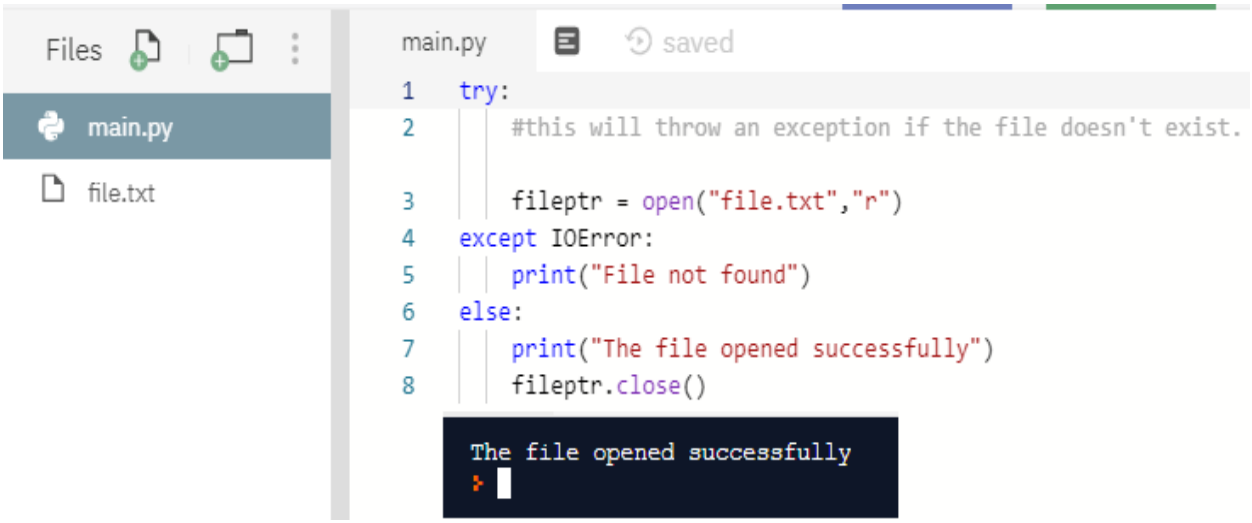
```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except Exception:
    print("can't divide by zero"
else:
    print("Hi I am else block")
```

```
Enter a:2
Enter b:0
can't divide by zero
```

```
Enter a:12
Enter b:2
a/b = 6
Hi I am else block
```

If file not present:

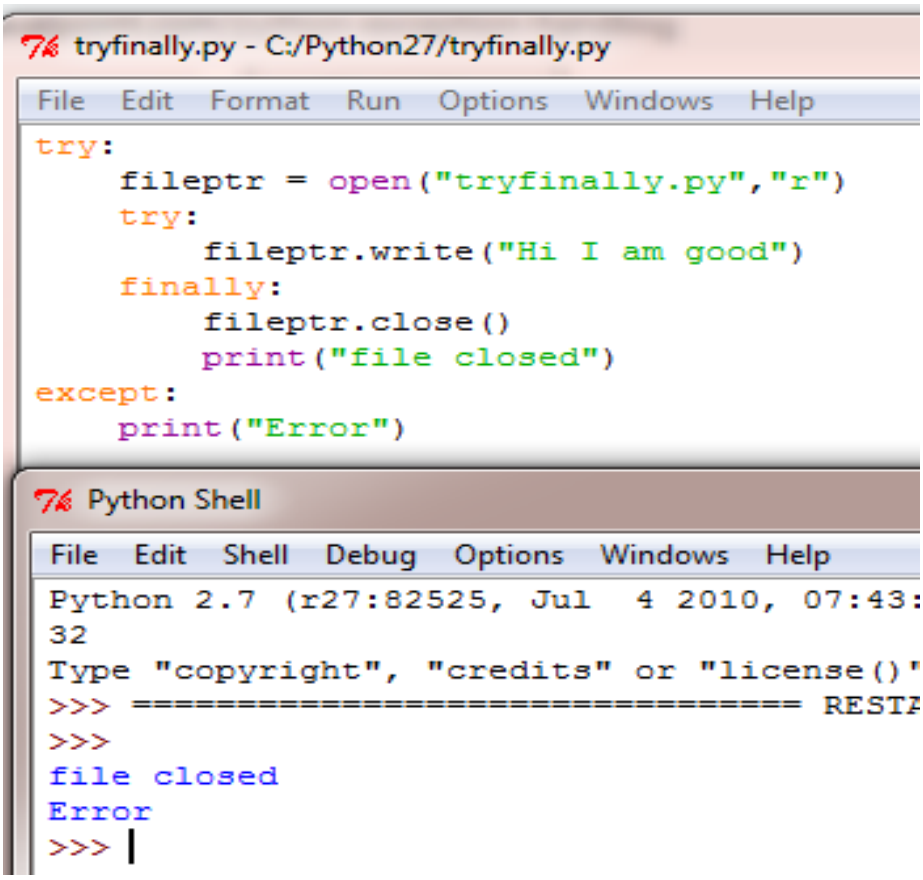
If file present:



The finally block

The finally block with the try block in which, we can pace the important code which must be executed before the try statement throws an exception.

(in given example we open file which is present in directory or filename where writing code.)



Raise an exception

To throw (or raise) an exception, use the raise keyword.

main.py saved

```
1  try:
2      age = int(input("Enter the age?"))
3      if age<18:
4          raise ValueError;
5      else:
6          print("the age is valid")
7  except ValueError:
8      print("The age is not valid")
```

```
Enter the age?1
The age is not valid
```

User defined exceptions.

Python has many built-in exceptions which forces program to output an error when something in it goes wrong.

However, sometimes need to create a custom exception that serves purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class.

main.py saving...

```
1   class invalidPassword(Exception):
2     pass
3   def verify(pwd):
4     if str(pwd)!="abc":
5       raise invalidPassword
6     else:
7       print("valid Password")
8
9   verify("abc")
10  print("\n")
11  print("now see ex when raise exception\n\n")
12  verify("pwd")
```

```
valid Password


now see ex when raise exception


Traceback (most recent call last):
  File "main.py", line 12, in <module>
    verify("pwd")
  File "main.py", line 5, in verify
    raise invalidPassword
__main__.invalidPassword
```